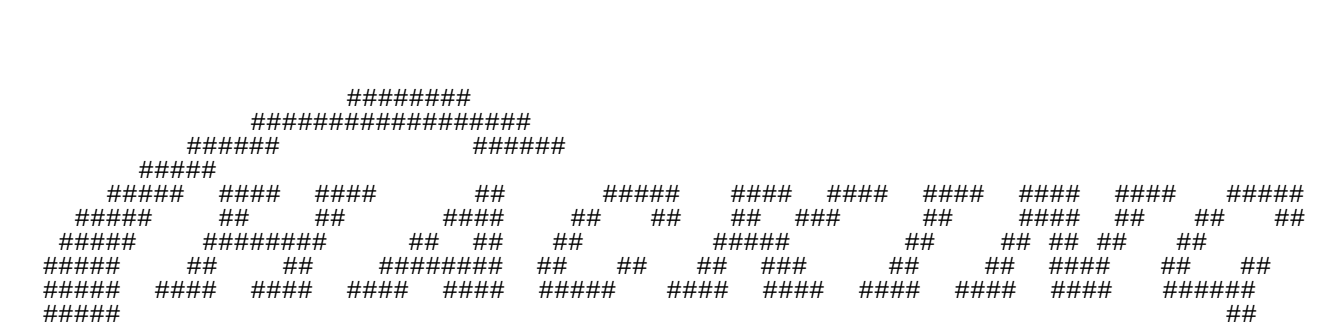


Issue #10
June 30, 1995

Editor's Notes
by Craig Taylor

This is my last issue of Commodore Hacking (having finally gotten out the door, but I couldn't break tradition and get it out on time :-)). I'm having to give it up because I've gradually lost interest in Commodore computers over the years and with the search for a job (anyone wanna hire a csc graduate?) and as I get older I seem to have less and less time.

I'm gonna be handing the reigns of Commodore Hacking over to Jim Brain, who is a very active member of the Commodore Internet community. He will also be running a mailserver that will take the place of mine (Mine will become unavailable after July 1st and will send pointers to Jim Brain's mailserver).

It's been interesting to watch the Commodore computers evolve, take off like a rocket and then have Commodore go into liquidation. Commodore computers have been and still are, (with some exceptions - 1541 head-banging comes to mind), technologically sound. For a "hacking" machine they're wonderful.

My email address has changed to duck@nando.net. I periodically still check mail at duck@pembvax1.pembroke.edu but only every 2 weeks or so. I am still going to try to be in the Commodore community but time will govern my ability to do that. I'm going to miss editing this rag....

And here is Jim Brain:

Mail Server Changes:

With Issue 10, the address for the Commodore Hacking mail server has changed. The new address is brain@mail.msen.com The commands are the same as before. Not all of the files have been moved yet, so please email the administrator (Jim Brain, brain@mail.msen.com) if a file you need is not on the new site.

Howdy:
Howdy, my name is Jim Brain, and I will be taking over the position of editor for Commodore Hacking starting with Issue 11. Some of you may know me as the Commodore Trivia Contest administrator, the USENET newsgroup comp.sys.cbm FAQ Manitainer, or the keeper of a Commodore Information WWW Site at http://www.msen.com/~brain/cbmhome.html. Wherever you have heard of me from, or even if you haven't, I will state that I plan on handling Commodore Hacking in the following way. The next issue will possibly look different cosmetically, as I edit somewhat differently than Craig, but the content and basic layout will remain the same. The types of material will not change, and the structure for submitting articles will change only in the address to mail them to: brain@mail.msen.com. However, I do have a few changes in mind:

1) Try to stabilize the issue generation so that Commodore Hacking will become a quarterly publication.

2) Attempt a fully "HTML"ized version of the magazine, while still providing a text version.

3) Pursue the possibility of providing a printed version of these issues for those who have no online access to them.

4) Encourage User Groups and other CBM related organizations to carry this magazine for their members.

So, again I say howdy. As always, Commodore Hacking will accept your articles at any time, so please help us keep this quality magazine running. If you have any questions or comments about the change in editorship, the possible changes, or other matters, please feel free to drop me a note. I look forward to hearing from you and publishing your articles.

Jim Brain
brain@mail.msen.com

==========

Legal Mumbo-Jumbo

Permission is granted to re-distribute this "net-magazine", in whole, freely for non-profit use. However, please contact individual authors for permission to publish or re-distribute articles separately. A charge of no greater than 5 US dollars or equivlent may be charged for library service / diskettte costs for this "net-magazine".

Please note that this issue and prior ones are available via anonymous FTP from ccnga.uwaterloo.ca (among others) under /pub/cbm/hacking.mag and via a mailserver which documentation can be obtained by sending mail to "brain@mail.msen.com" with a subject line of "mailserver" and the lines of "help" and "catalog" in the body of the message.

In This Issue:

Commodore Trivia

Trivia Edition #13-18 are in this article. As you may know, these questions form part of a contest in which the monthly winner gets a prize (Thanks to my various prize donators). The whole thing is mainly just for fun, so please enjoy. Try your hand at Commodore trivia!!

BFLI - New graphics modes 2

FLI gave us more color to the screen, AFLI increased the horizontal resolution and color selection by using the hires mode. BFLI stands for 'Big FLI' and gives us 400 lines instead of the usual two hundred. AFLI and BFLI can be combined, but we are not going into that.

Making stable raster routines (C64 and VIC-20)

In this article, I document two methods of creating stable raster routines on Commodore computers. The principles apply for most 8-bit computers, not only Commodores, but raster effects are very rarely seen on other computers.

A Differant Perspective - Part III.

Yes!!! It's yet another article on 3D graphics! Even if you haven't been following this series, you can use this program. This time around we will write a completely general polygon plotter -- if you can type basic data statements, you can create a three-dimensional object out of polygons and rotate and project it to your heart's content. For the more technically inclined we will look at optimizations to the line routine, EOR-buffer filling, and more! Yow!

Second SID Chip Installation

This article describes how to add a second sid chip for use in SidPlayer and other programs. As always, be extra careful when making modifications to your computer.

Solving Large Systems of Linear Equations on a C64 Without Memory

OK, now that I have your attention, I lied. You can't solve dense linear systems of equations by direct methods without using memory to store the problem data. However, I'll come back to this memory free assertion later. The main purpose of this article is to rescue a usefull numerical algorithm, "Quartersolve", and also to provide a brief look at the COMAL programming language and BLAS routines.

The World of IRC - A New Life for the C64/128

I've heard people talking about IRC. What is it? Why is it useful to me as a Commodore user? Bill "Coolhand" Lueck explains the hows and whys in this article.

SwiftLink-232 Application Notes (version 1.0b)

This information is made available from a paper document published by CMD, with CMD's permission.

Design and Implementation of a Simple/Efficient Upload/Download Protocol

This article details how to implement a custom upload/download protocol that is faster than most of the ones common to the C64/128 computers.

Design and Implementation of a 'Real' Operating System for the 128: Part II

There has been a slight change in plans. I originally intended this article to give the design of a theoretical distributed multitasking microkernel operating systemfor the C128. I have decided to go a different route: to take out the distributed component for now and implement a real multitasking microkernel OS for a single machine and extend the system to be distributed later. The implementation so far is, of course, only in the prototype stage and the application for it is only a demo. Part III of this series will extend this demo system into, perhaps, a usable distributed operating system.

==========

Trivia
by Jim Brain (brain@mail.msen.com)

Well, summer is upon the Brain household, and things are moving at a fast clip at the house. However, the trivia still keeps coming. I appreciate all the people who contribute to the trivia and all the people who take part in the monthly contest. I have collected Trivia Edition #13-18 in this article. As you may know, these questions form part of a contest in which the monthly winner gets a prize (Thanks to my various prize donators). The whole thing is mainly just for fun, so please enjoy.

As the summer months start up, news on the trivia includes:

1) I now have access to some more orphan machines (C65, C116), so expect some trivia questions on those models.

2) The new home now has a number of machines set up, so testing answers to the trivia is even easier. I am still trying to get the old PET machines in house, but the others are here.

3) The Commodore World Wide Web Pages (http://www.msen.com/~brain/cbmhome.html) that I maintain and place the trivia on caught the eye of USA Today and the Pheonix Gazette. I was interviewed for both articles. Look in the June 20th edition of USA Today for the segment, and possibly a picture of Jim Brain and the machines he uses to create the trivia.

As always, I welcome any questions (with answers), and encourage people to enter their responses to the trivia, now at #18.

Jim.

The following article contains the answers to the December edition of trivia ($0C0 - $0CF), the questions and answers for January ($0D0 - $0DF), February ($0E0 - $0EF), March ($0F0 - $0FF), April ($100 - $10F), and the questions for the May edition ($110 - $11F). Enjoy them!

Here are the answers to Commodore Trivia Edition #13 for December, 1994

Q $0C0) The early 1541 drives used a mechanism developed by _____. Name the company.

A $0C0) Alps.

Q $0C1) On later models, Commodore subsequently changed manufacturers for the 1541 drive mechanism. Name the new manufacturer.

A $0C1) Newtronics.

Q $0C2) What is the most obvious difference(s). (Only one difference is necessary)

A $0C2) Alps: push-type latch, round LED.
Newtronics: lever-type latch, rectangular LED.

Q $0C3) On Commodore BASIC V2.0, what answer does the following give:
PRINT (SQR(9)=3)

A $0C3) 0. According to Commodore BASIC, the answer should bby -1, which is the BASIC value of TRUE. However, the above equation is NOT true. Doing PRINT SQR(9) yields 3, but doing PRINT (SQR(9)-3) yields 9.31322575E-10 (C64). This anomaly can be attributed to roundoff errors in the floating point math routines in Commodore BASIC.

Q $0C4) In Commodore BASIC (Any version) what does B equal after the following runs: C=0:B=C=0

A $0C4) B = -1. The second statement is the one to look at. The second equals sign is treated as a comparison, while the first is treated as a assignment. B gets set to the outcome of the comparison, which is TRUE (-1).

Q $0C5) The first PET cassette decks were actually _____ brand cassette players, modified for the PET computers. Name the comapny.

A $0C5) Sanyo. Specifically, Model M1540A. What a model number!

Q $0C6) In Commodore BASIC (Any version), what happens if the following program is run:

```
10 J=0
20 IF J=0 GO TO 40
30 PRINT "J<>0"
40 PRINT "J=0"
```

A $0C6) On BASIC 2.0 or greater:

```
?SYNTAX  ERROR IN 20
READY.
```

On BASIC 1.0: (found on the PET 2001 series)

```
J=0
READY.
```

BASIC 1.0 totally ignored spaces, so line 20 became "IFJ=0GOTO40". That statement would be correctly parsed, sicne it contains the "GOTO" keyword.

However, on BASIC 2.0 or greater, spaces weren't ignored so completely, and the "TO" in "GO TO" would be tokenized separately, so some code was added to BASIC to check to "GO". As the code that accepts GOTO as a special case for THEN after an IF statement wasn't patched this way, the above fails, because GO is not a valid keyword after IF. The statement SHOULD work correctly, but does not because of this failure to fix the IF command parsing.

On BASIC 2.0 or greater, substituting the following line for line 20 will cause the program to work:

```
20 IF J=0 THEN GO TO 40
```

Q $0C7) In question $068, we learned how Jack Tramiel first happened upon the name "COMMODORE". According to the story, though, in what country was he in when he first saw it?

A $0C7) Germany.

Q $0C8) On the Commodore user port connector, how many edge contacts are there?

A $0C8) 24. Two rows of 12 contacts each.

Q $0C9) On most Commodore computers, a logical BASIC screen line can contain up to 80 characters. On what Commodore computer(s) is this not true?

A $0C9) According to Commodore documentation, a _physical_ screen line is defined as one screen line of characters. A _logical_ screen line is defined as how many _physical_ lines can be chained together to create a valid BASIC program line.

With that in mind, most Commodore computers chose a _logical_ screen line that was a multiple of the screen width. This works fine for 40 and 80 column screens, but what do we do with the VIC-20, with its 22 column screen. Solution: make the _logical_ line length equal to 4 _physical_ lines, or 88 columns.

When the Commdore 128 was introduced, the number rose to 160 characters, which is 4 _physical_ lines in 40 column mode, or 2 _physical_ lines in 80 column mode. However, you can only take advantage of this in 128 mode. 64 mode is limited to 80 characters.

To add to all this confusion, a valid BASIC program line (in memory) can actually be 255 (tokenized) characters long, but creating such

a long line cannot be done from the built-in editor in direct mode.

The AmigaBASIC, available on the Amiga, also does not have the 80 column line limit. However, that BASIC is SOOO much different that I am not surprised. The older CBM BASICs, on the other hand, were all derivatives of the original Level 1 BASIC for the PET.

Q $0CA) If a file is saved to a Commodore Disk Drive with the following characters: chr$(65);chr$(160);chr$(66), what will the directory entry look like?

A $0CA) The filename will show up as "A"B, with the 'B' showing up to the right of the '"' mark. This could be used to make program loading easier. A file that showed up as "filename",8,1 could be loaded by simply hitting shift-run/stop on that line.

Q $0CB) What is the maximum length (in characters) of a CBM datasette filename?

A $0CB) References I have on hand say 128 characters. However, the actual code on the 8032 and the C64 acts as though 187 characters can actually be sent (tape buffer-5 control bytes = 192-5=187). The references that claim 128 characters are Nick Hampshire's _The VIC Revealed_ and _The PET Revealed_. ANyone care to lay this one to rest?

Q $0CC) How many keys are on a stock Commodore 64 keyboard?

A $0CC) 66 keys. This is the same number as found on the VIC-20 and the Commodore 16.

Q $0CD) Commodore BASIC uses keyword "tokens" to save program space. Token 129 becomes "FOR". What two tokens expand to include a left parenthesis as well as a BASIC keyword?

A $0CD) TAB((163) and SPC((166).

Q $0CE) There are 6 wires in the Commodore serial bus. Name the 6 wires.

A $0CE) 1) Serial /SRQIN
2) GND
3) Serial ATN IN/OUT
4) Serial CLK IN/OUT
5) Serial DATA IN/OUT
6) /RESET

Q $0CF) On the Commodore datasette connector, how many logical connections are there?

A $0CF) 6. Opposing pins on the connector are hooked together electrically.

Here are the answers to Commodore Trivia Edition #14 for January, 1995

Q $0D0) How many keys were there on the "original" PET and what was special about them?

A $0D0) the original PET had 73 calculator-style keys that were laid out in a rectangular matrix, not typewriter-style.

Q $0D1) How do you produce the "hidden" message(s) on the Commodore 128?

A $0D1) SYS 32800,123,45,6. The screen will clear, and the software and hardware developers on the 128 project will be named.

The exact text is as follows:

[RVS] Brought to you by...

Software:
Fred Bowen
Terry Ryan
Von Ertwine

Herdware:
Bil Herd
Dave Haynie
Frank Palaia

[RVS]Link arms,don't make them.

Q $0D2) How much memory did the "original" PET show on bootup?

A $0D2) The "original" PET came in two configurations, 4K and 8K, so:

  The PET 2001-4 had 3071 bytes.
  The PET 2001-8 had 7167 bytes.

Q $0D3) We all know the "reboot" sys for the 64 is sys 64738, but who knows the same sys location to reboot the CBM 8032?

A $0D3) sys 64790

Q $0D4) Which computer(s) beeped at bootup? (May be more than one, but only one required)

A $0D4) I know some of these are corect, but the sheer size of the list prevents me from checking them ALL out.

FAT 40XX series
80XX series
PC-10 (I suspect a number of IBM clones did, and these things have no consistent naming convention across country boundaries.)
PC-20
Amiga 1000
SP9000 (SuperPET)

Q $0D5) How much memory did the CBM 8032 show on bootup?

A $0D5) 31743 bytes.

Q $0D6) Certain Commodore computers provided emtpy EPROM sockets on the motherboard. Give me the number of empty sockets on the following machines:

a) CBM 30XX.
b) CBM 8XXX.
c) CBM C128.
d) Plus/4.

A $0D6) a) 3 sockets.
b) 2 sockets.
c) 1 socket.
d) 1 socket.

Q $0D7) In Germany, the CBM 8032 came with a 4kB EPROM for the EXXX area, while the US version only had a 2kB EPROM. Why?

A $0D7) The German version had additional keybaord drivers for umlaut characters and dead keys.

Q $0D8) Who published the first PET memory map in the "PET Gazette"?

A $0D8) None other than the infamous Jim Butterfield.

Q $0D9) Which is faster to move the sursor on a PET/CBM or C64: SYS or PRINT?

A $0D9) PRINT is faster, since the sys approach must process the pokes before the sys, which are very slow.

Q $0DA) On the Amiga 1000, where are the signatures of the first Amiga developers located?

A $0DA) Inside the top case of the Amiga (1000).

There is an interesting footnote to this question. It seems that at least some original Amiga machines were labeled as Amiga (with nu number). Then, at some later point, the number was added. In addition, Commodore produced some Amiga 1000 machines without the signatures, but most had the telltale handwriting on the inside of the case.

Q $0DB) On the 6502, what does the accumulator contain after the following is executed:

```
lda #$aa
sed
adc #01
```

A $0DB) Assume carry was clear. If so, then $11 is the correct answer.

Q $0DC) What is the model number of the US NTSC VIC-II chip?

A $0DC) Its first number was 6567, and that is the number most people know it by, but Commodore produced a VIC-II using a new manufacturing process that was numbered the 8562.

Q $0DD) What is the European PAL VIC-II chip's model number? (Not sure if that's its rightful term, but I hope you understand).

A $0DD) Same here. The part number 6569 is the most remembered number, but an 8565 will work as well.

Q $0DE) Assume you have two computers, one with each of the above chips inside. Which chip draws more pixels on the screen per second?

A $0DE) Note, for the purposes of the calculation I am performing, "pixels" refers to picture elements that can be adddress and modified using normal VIC modes, so there are 320*200 "pixels" on both the PAL and NTSC screens. (I probably should have stated this, but it is too late now.) Also, the screen refresh rates used in the calculations are those defined by the respective television standards (60Hz U.S., 50Hz European), even though the actual frequencies are off by a small percentage. (for example, the actual 50Hz refresh rate on European VIC-II chips was calculates as 50.124567Hz by Andreas Boose)

So, the PAL draws 320*200*50 pixels per second = 3200000 pixels/s
NTSC draws 320*200*60 pixels per second = 3840000 pixles/s

Now, some people thought I meant the whole screen, not just the display area provided by the VIC-II chip. Well, I am not sure exactly you calculate pixels on a screen, since the numbers could vary from display to display, but if we measure in scanlines:

PAL = 312 scanlines * 50 = 15600 scanlines/s
NTSC = 262 scanlines * 60 = 15720 scanlines/s

The NTSC machines wins both ways.

Q $0DF) In Commodore BASIC, which statement executes faster:

```
a = 2--2
```

or

```
a = 2+2
```

A $0DF) b is the correct answer, and there are a couple of reasons why:

1) 2--2 takes longer to parse in the BASIC interpreter.
2) Commodore BASIC subtracts by complementing the sign of the second number and adding. This incurs extra time.

There are even more subtle ones, but I leave them as an exercise for the reader. Send me your reason why.

Here are the answers to Commodore Trivia Edition #15 for February, 1995

Q $0E0) What is the difference(s) between the Newtronics 1541 and the 1541C? (only one difference is needed)

A $0E0) (George Page, a noted authority on CBM Drives, indicated that Commodore made this a tough question to answer.) By the time the 1541C was introduced, Commodore threw a number of drives together and called them 1541Cs. The theoretical 1541C exhibited the following features:

No head banging, and other problems fixed by modified ROMs.
Case color matches C64C and C128 computers.

Q $0E1) What happens when you type 35072121 in direct mode on the C64 and hit return?

A $0E1) Simple answer: Most likely, the screen clears and the word READY. is printed at screen top. This is the behavior seen when pressing RUN-STOP/RESTORE. Alternately, nothing could happen, or the computer could lock up.

Involved answer: There is a bug in BASIC 2.0. Easily fixed, but destined to live life immortal. (long)

The bug is in the PETSCII number to binary conversion routine at $a69b (LINGET). The routine basically reads in a character from the line, multiplies a partial result by 10 and adds the new character to the partial result. Here is a code snippet:

```
a96a     rts
a96b     ldx #$00  ; zero out partial result
a96d     stx $14
a96f     stx $15
a971     bcs $a96a ; not a number, return
a973     sbc #$2f  ; PETSCII to binary
a975     sta $07
a977     lda $15   ; get hi byte or partial result
a979     sta $22
a97b     cmp #$19  ; partial > 6399
a97d     bcs $a953 ; yes, goto error
a97f     lda $14   ; load lo byte of result
a981     asl       ; lo*2
a982     rol $22   ; hi*2 + c
a984     asl       ; lo*2
a985     rol $22   ; hi*2 + c
a987     adc $14   ; complete lo*5
a989     sta $14
a98b     lda $22
a98d     adc $15   ; complete hi*5
a98f     sta $15
a991     asl $14   ; lo*2 complete lo*10
a993     rol $15   ; hi*2 complete hi*10
a995     lda $14
a997     adc $07   ; add new char
a999     sta $14
a99b     bcc $a99f ; did lo overflow?
a99d     inc $15   ; yes, inc hi
a99f     jsr $0073 ; get next char
a9a2     jmp $a971 ; go through it again.
```

The problem is at $a97d. when the partial result is greater than 6399, (if partial > 6399, then new partial result will be over 63999) the routine needs to get to $af08 to print an error, but can't due to branch restrictions. However, a branch that will get there is in the preceding function, which handles the ON GOTO/GOSUB keywords ($a94b, ONGOTO).

So, the BASIC writers just branched to the code in ONGOTO; specifically $a953:

```
a94b     jsr $b79e
a94e     pha
a94f     cmp #$8d  ; is the keyword GOSUB ($8d)
a951     beq $a957 ; yes
a953     cmp #$89  ; is the keyword GOTO ($89)
a955     bne $a8e8 ; no, print SYNTAX ERROR.
a957     ...       ; handle ON GOTO/GOSUB
```

This code is checking to make sure the ON (var) is followed with a GOTO or GOSUB keyword.

The LINGET error handler branches to $a953, which compares .A (which holds hi byte of partial result) to $89. Normally, this fails, and the normal SYNTAX ERROR code is reached through the branch to $a8e8. However, for partial results of the form $89XX, the check succeeds, and BASIC tries to execute an ON GOTO/GOSUB call.

By the way, it is no coincidence that this error occurs on 35072121, since one of the partial results is $8900 (hi byte is $89). In fact, 350721 will achieve the same result.

If the check succeeds, the code limps along until $a96a:

```
a969     pla       ; complement to $a94e
a96a     rts       ; return
```

But we never executed $a94e, the push, so the stack is now messed up. Since the stack held $9e, $79, $a5 before the PLA, (The stack could hold other values, but I always saw these) the RTS gets address $a579 to return to, which usually holds a BRK opcode. The break handler is invoked, and the screen clears with the READY. at the top.

Now, the BASIC 2.0 authors were justified in reusing the error

handler code in ONGOTO for LINGET, but they calculated the branch offset wrong, according to my tests. If you have the LINGET error handler branch to $a955, all these troubles disappear. You can verify this procedure with the following BASIC program on a 64:

```
10 for t=57344 to 65535:poke t,peek(t):next
20 for t=40960 to 49151:poke t,peek(t):next
30 poke 43390, 214
40 poke 1, peek(1) and 254
```

Just to be complete, this error occurs when a 6 digit or greater line number is entered and the first 6 digits indicate a number in the range 35072-35327 ($8900-$89ff). Also, it appears the error occurs on the VIC-20, but I didn't completely verify it. It would be interesting to note if the error is found on all version of CBM BASIC.

Whew, what a mouthful.

Q $0E2) If a SID chip is producing a "sawtooth waveform", does the waveform look like:

a) "/|/|/|/|" or
b) "|\|\|\|\" ?

A $0E2) a is the correct answer.

Q $0E3) On BASIC 2.0, what special precaution(s) must one take when working with relative files? (only one is needed)

A $0E3) Because BASIC 2.0 doesn't handle positioning in relative files quite right, one must position the relative file pointer before AND AFTER a read or write to a relative file.

Q $0E4) What incompatibility existed between C128 Rev. 0 ROMS and the REU?

A $0E4) OK, I admit it. I placed this answer and its discussion somewhere in my store of information, and it must have fallen behind the cabinet, because I cannot find it. I will post an answer to this as soon as I can find it, but the answers really must go out, as they have been held up long enough.

Q $0E5) What can trigger an NMI interrupt? (count all sources on one chip as one)

A $0E5) The following sources can trigger an NMI interrupt:

1) The expansion port.
2) CIA #2.
3) The RESTORE key.

Q $0E6) What can trigger an IRQ interrupt? (count all sources on one chip as one)

A $0E6) The following sources can trigger an IRQ interrupt:

1) The VIC-II chip.
2) CIA #1.
3) The expansion port.

Q $0E7) Where is the ROM in a 1541 located in the 64K memory map?

A $0E7) The ROM is located from $C000 to $FFFF, yet the ROM code does not begin until $C100.

Q $0E8) Which VIA on the 1541 is hooked to the read/write head?

A $0E8) VIA #2, located in memory from $1C00 to $1C0E.

Q $0E9) In the Commodore DOS, what bit in the file type byte denotes a "locked" file?

A $0E9) bit 6.

Q $0EA) If files are "locked" under Commodore DOS, under what condition(s) may the file be changed?

A $0EA) Depending on the file, the following operations can be done on a locked file:

1) Rename will change file name, although not contents of file.
2) Random access can be used to alter file.

3) Formatting the disk will alter the file. (duh!)
4) Save-with-replace (@0:) will replace file and unlock it.
5) Opening file in append mode will allow it to be changed, and unlock it.
6) Opening a relative file and adding or changing a record will succeed and unlock file.

Q $0EB) How big can a program file be on a 1541 or similar?

A $0EB) The file can be as large as a sequential file, since both are stored in the same way: 168656 bytes. However, since a program contains its load address as bytes 0 and 1, the largest program size is 168654 bytes.

Q $0EC) Under BASIC 2.0, how does one open a random access file on a disk drive?

A $0EC) Random access (or direct access) files are a misnomer. What you really doing is opening the disk for reading and writing. You need two open command to access a random file: (assume drive 8)

open 15,8,15 and

open 1,8,4,"#1" will open a random access file using buffer 1.
open 1,8,4,"#" will open a random access file using the first available buffer

Now, by using B-R, B-W, B-A or their replacements, you can write data to sectors on the disk.

Note that Random access files are different from relative files.

Q $0ED) A file that has a '*' immediately before the filetype is called a _____ file.

A $0ED) a splat file. This is its correct term, believe it or not.

Q $0EE) We know the 1541 and similar drives have 5 internal buffer areas, but how many does an 8050 drive have?

A $0EE) Since the 8050 has twice the on-board RAM (4kB), it has 16 buffers, but only 13 are available. (All CBM drives use one buffer for zero-page memory, one for stack memory, and one for temporary variables.)

Q $0EF) On a "save-with-replace", where is the location of the first track and sector of the new copy of the program saved in the directory entry for the old copy?

A $0EF) The new first track is stored at location 26, and the new first sector is stored at location 27. These values are copied to their correct locations after the save is completed.

Here are the answers to Commodore Trivia Edition #16 for March, 1995

Q $0F0) What size matrix of pixels comprises a character on a PET 2001 computer?

A $0F0) The matrix was 8 by 8.

Q $0F1) How many bytes did the opening screen on a CBM 4016 show as available for use by BASIC?

A $0F1) 15359 bytes free.

Q $0F2) The character set that produces uppercase letters on unshifted keys is the _____ character set.

A $0F2) "standard mode".

Q $0F3) The character set that produces lowercase letters on unshifted keys is the _____ character set.

A $0F3) "alternate mode"

Q $0F4) To get to the set mentioned in $F2, what character code would be printed to the screen?

A $0F4) chr$(142)

Q $0F5) What character code would one print to the screen to invoke the

chararacter set in $F3?

A $0F5) chr$(14)

Q $0F6) If one does LIST 60-100, will line 100 get "listed"?

A $0F6) Yes. The above translates as: LIST 60 through to and including 100.

Q $0F7) The abbreviation for the BASIC 4.0 command "COLLECT" is _____.

A $0F7) coL. "C" "O" "SHIFT-L". For those who are interested, the COLLECT command is analogous to the VALIDATE operation.

Q $0F8) When you use a subscripted variable in BASIC, how many elements are created by default if no DIM statement is issued?

A $0F8) 11 elements. A(0) - A(10). Almost everyone who has ever programmed in Commodore BASIC has seen the "BAD SUBSCRIPT" error when they try to use the 12th element in a un-DIMensioned array.

Q $0F9) How large is the keyboard buffer in CBM computers?

A $0F9) 10 bytes. Since this area could be POKEd to, many boot programs would poke characters into this buffer to simulate keypresses.

Q $0FA) On the Commodore 1581, how large is a physical sector in bytes?

A $0FA) A physical sector is 512 bytes in length. Internally, the 1581 creates 2 256 "logical" sectors in a physical sector, to maintain compatibility with older Commodore drives.

Q $0FB) You'll find BASIC 3.5 on the _____ line of CBM computers.

A $0FB) The X64 series. That includes the Commodore 16, the Commodore 116, and the Commodore Plus/4.

Q $0FC) On the Commodore 1351 mouse, what registers in the Commodore computer would the X and Y proportional information be read from?

A $0FC) Even though you are looking for digital information (how far the mouse has traveled since the last movement in a particular axis), the information is read from the "paddle" or potentiometer (POT) registers. On the C64, the POT registers are part of the SID chip, and are at 54297 ($D419) for POTX, and 54298 ($D41A) for POTY.

Q $0FD) What is the maximum size of a sequential file on a 1581 drive?

A $0FD) 802640 bytes.

Q $0FE) What flaw exists in the early Commodore 1670 modems?

A $0FE) When the 1670 modem was first introduced, it powered up in auto-answer mode, which means it would answer incoming calls after the phong rang. You could turn this feature off through software control, but if the power was reset, the modem would answer the phone. So many people complained to Commodore that CBM revised the 1670 to include an extra DIP switch that turned this feature off.

Q $0FF) What is the model number of the first modem for the VIC and C64?

A $0FF) The 1600 manual dial/manual answer 0-300 bps modem. The author owns one, and used it for many years. To operate, you must use a phone with a detachable handset cord. You dialed the number on the phone, waited for the answer, unplugged the handset, and plugged the cord into the 1600. A switch toggled between using originate or answer frequencies. The 1600 was manufactured by Anchor Automation for Commodore. (As an aside, this unit claimed 300 bps, but I never could get 300 to work well. Most of my telecommunications happened at 150 bps.)

-------Commodore Trivia Edition #17 Questions and Answers (BEGIN)--------

Q $100) On the MOS Technology's KIM-1, how many keys were on the keypad?

A $100) 23 keys. The keypad has room for 24, but one spot is taken by a switch that puts the system into single-step mode. Interestingly, some pictures have the switch on the upper left, some on the upper

right.

Q $101) The KIM-1 keypad had the common 0-9A-F keys on the keypad, but also had some special keys. Name them.

A $101) GO (Go) Executes an instruction and displays the address of next,
ST (Stop) Stops execution of program and return control to monitor,
RS (Reset),
AD (Address) Address entry mode,
DA (Data) Data entry mode,
PC (Program Counter) Displays and restores program counter to values in PCL and PCH,
+ (Increment) Increments the address without changing the entry mode.

Q $102) The KIM-1 was a set of modules that could be plugged together to expand the system. Each module had a model number. What was the model number of the KIM-1 motherboard?

A $102) The KIM-4.

Q $103) On the 1525 line of printers, if you wanted to create the following graphic, what bytes would you send to the printer after turning on graphics mode?

```
****
*  *
*  *
*  *
*  *
*  *
****
```

A $103) I guess I should have stipulated that this is a bitmap. ASCII just has a few limitations. Anyway, the correct bytes to send are: 255, 193, 193, 255. You got these by assigning each bit in a column a value, and adding 128 to the result for each column.

Q $104) What is the horizontal resolution of the 1525 line of printers?

A $104) Character resolution: 80 chars, or 10 chars/inch (cpi).
Graphics resolution: 480 dots, or 60 dots/inch (dpi).

Q $105) On Commodore drives, explain the difference between the B-R command and the U1 command.

A $105) The two commands read in data from a disk sector. However, the U1 command always reads a full sector (255 bytes). The B-R command reads the number of bytes specified in the first byte of the sector. If the first byte is a 15, B-R will read 15 bytes from the sector. (From the 1581 manual)

Q $106) On the Commodore 1541 drive, what does the U: command do?

A $106) This command has been traditionally used to reset Commodore drives, including the CBM 1541. However, some early versions of the Drive DOS did not correctly handle this command. In these versions, the drive and computer failed to complete the command transaction successfully, and what looked like a hung machine resulted. Commodore later fixed this problem. If U: seems to not work on your drive, try U; instead.

Q $107) What does the first routine in the 1541 drive ROM actually do?

A $107) The function, called SETLDA and residing at $C100, turns on the drive active LED for the current drive. The routine loads the current drive from $7F and sets bit 3 of DSKCNT ($1C00).

Q $108) How many files will a 1581 disk drive hold?

A $108) 296 files. Note that it is not a multiple of 144.

Q $109) Commodore 1581 drives have a special "autoboot" feature that enables the drive to load and run a program off a disk upon drive bootup. What is the required name of the file?

A $109) COPYRIGHT CBM 86

Q $10A) What filetype must the file mentioned in $109 be?

A $10A) USR.

Q $10B) To power up a 1351 mouse in "joystick mode", what must the user do?

A $10B) If one depresses the right mouse button during power-up, the 1351 will behave just like a joystick.

Q $10C) Describe the contents of the POTX or POTY registers when using a 1351 mouse.

A $10C) Each register holds the same type of information, just for a separate axis, so we will describe just one register:

Bit:	Function
7	Don't care
6-1	Mouse axis position mod 64.
0	Noise Bit. (check this bit to see whether mouse has moved)

Q $10D) Commodore computers typically use most of zero page for temporary variables and other items. However, both the VIC-20 and the 64 reserve 4 bytes for user programs that need zero page memory. Where are these locations?

A $10D) $FB-$FE (251-254). I am not sure these were "reserved" for programmers as much as they were just not utilized by the CBM programmers.

Q $10E) Name the 16 colors available on the 64.

A $10E) Black
White
Red
Cyan (Light Blue-Green)
Purple
Green
Blue
Yellow
Orange
Brown
Light Red
Dark Gray (Gray 1)
Medium Grey (Gray 2)
Light Green
Light Blue
Light Gray (Gray 3)

Q $10F) Both the VIC-20 and the C64 emulate the operation of the 6551 UART. How many "mock 6551" registers are mapped into the memory map?

A $10F) 5, from $293-$297 (659-663). The register contents:

$293	6551 Control Register
$294	6551 Command Register
$295-6	6551 User Defined Baud Rate value.
$297	6551 Status Register

----------Commodore Trivia Edition #18 Questions (BEGIN)----------

Q $110) What is the name of the company that recently purchased the liquidated Commodore assets?

Q $111) At one time, Commodore attempted to manufacture a dual drive version of the 1571 called the 1572. For what technical reason did it utimately fail?

Q $112) Over what computer system did a User Group sue Commodore and win?

Q $113) In $103, the question asked how to create a graphic of a small box on the 1525. In this quesrtion, we have made a different design. If you wanted to create the following graphic using individual dots on the printer, what bytes would you send to the printer after turning on graphics mode?

```
 **     * *
*       ***
*   **  ***
*   * * * *
 ** **  * *
    * *
    **
```

Q $114) (Some C65 questions) How many SID chips does the the development
        Commodore 65 machine contain?

Q $115) What CPU does the Commodore 65 use?

Q $116) What is the alternate name for the Commodore 65?

Q $117) How many processors does the internal 1581-compatible drive
        on the C65 contain?

Q $118) In the tradition of naming certian ICs after famous cartoon
        characters, one of the ICs in the C65 is named after a Warner
        Brothers cartoon character. Which one?

Q $119) What version of BASIC is included on the Commodore 65 in C65 mode?

Q $11A) How many I/O ports does a Commodore 65 contain?

Q $11B) What common Commodore 64 I/O port does the C65 NOT have?

Q $11C) How many function keys are on a Commodore 65?

Q $11D) What CBM disk drive DOS was used as the template for the internal
        C65 drive DOS?

Q $11E) What resolution of text screen does the C65 power up in? (Please
        give answers in characters).

Q $11F) What distinguishing non-textual characteristic in the C65 is not
        present in othe Commodore 8-bit computers?

The information in this between the lines marked by (BEGIN) and (END)
is copyright 1995 by Jim Brain. Provided that the information
between the (BEGIN) and (END) lines is not changed except to correct
typographical errors, the so marked copyrighted information may be
reproduced in its entirety on other networks or in other mediums. For
more information about using this file, please contact the address
shown below.

Jim Brain
brain@mail.msen.com
602 North Lemen
Fenton, MI 48430
(810) 737-7300 x8528

Some are easy, some are hard, try your hand at:
    Commodore Trivia #18!

BFLI - New graphics modes 2

by Pasi 'Albert' Ojala <albert@cs.tut.fi>

One day I was watching some demos that used linecrunch routines for whole-screen multicolor-graphics upscrollers. I already had my theories about how and why linecrunch worked, but because I had not used it anywhere, the details were a bit vague. In fact, I have many times accidentally created linecrunch effects when trying to do something else with $D011. Probably every demo coder has.

But you learn by doing. I had the idea of using linecrunch for FLI instead of a simple multicolor picture as it always seemed to be used. However, this has probably been done before and because I don't like to do things that have been done before, I decided to use linecrunch to show a two-screen-tall FLI picture.

Linecrunch Basics

For those not familiar with linecrunch routines: linecrunch is used to scroll the screen UPWARDS by convincing VIC-II that it has already showed more character rows than it in reality has shown. Surprisingly (or then, maybe not :) this consists of fiddling with $D011. The timing is critical as always.

Linecrunch works by setting $D011 equal the line before the current line and VIC-II will happily think that it is time to move on to the next character row - add 40 to the video matrix counter, 320 to the graphics memory counter and be ready to start a bad line. Or, maybe 'NOT to go back to the current row' would be a more suitable description. (Programming VIC-II is slowly becoming a science.)

The required timing also does not cause bad lines so that you can

skip another line immediately on the successive line. In addition, lines can be skipped only after the first character row and half of the second character row have been displayed. This has something to do with the way VIC-II decides when there is a bad line.

Because linecrunch causes VIC-II to skip rows, it will run out of video matrix and color memory (and graphics memory) before reaching the end of the screen. However, VIC-II does not stop displaying the graphics nor does it reset the internal counters. The counters keep on running and wrap around instead.

Normally, when VIC-II is displaying the last character row, it is showing the memory from offsets $3c0 to $3e7. If VIC-II has skipped one character row, it is displaying from $3e8 to $40f instead. But, there are only 10 bits for the video matrix counter (0..1023), so it wraps around to zero after $3ff. This means that the beginning of the video matrix is displayed at the bottom of the screen. The character rows become shifted by 24 character positions to the right because there were originally 24 unused memory locations at the end of the memory (1000..1023). (To be honest, sprite image pointers are not unused memory, but they are not used with normal FLI.)

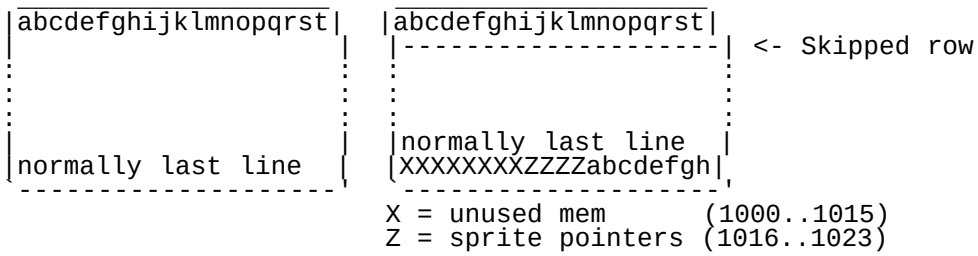


Figure 1: Linecrunch

The same thing happens for color memory because it uses the same counter for addressing the memory (in fact, color memory access and character data access are performed simultaneosly, 12 bits at a time). The graphics memory behaves the same way, except that the counter has three bits more and it counts at eight times the speed, so that it wraps at the exact same time as the other counter.

The first character row can't be used for linecrunch and the second one is also lost in the process. The first usable line to display is the third character row. However, those two lost rows can still be used as an extension at the end of the first screen. You must notice, however, that the alignment has been changed. After these two rows have been displayed, the video bank is switched to get new fresh data on the screen.

Back to BFLI

Wrapped data is nothing difficult to work with. It is just the matter of writing the right conversion program. Also, the normal FLI routine can be used, we just have to make sure VIC always has the right bank visible - simple LDA bank,x:sta $DD00 can accomplish that. The more difficult aspect is to make the display freely locatable. We have 32 kilobytes of graphics data, this is the main reason we can't even think about using copying. Linecrunch combined with the bad line delaying technique will do the job much more nicely.

Figure 2 shows the principles. To make things simpler I have chosen location 0 to mean that the top of the picture is visible, 1 means that the picture is scrolled one line upwards and so on. We can see that linecrunch is not used at all for the location 0. To make the picture start at the same point whether linecrunch has crunched lines or not we compensate the non-lost raster lines by delaying the next bad line. When the location is n*8 (n=0,1,2..), the sum of the linecrunched and delayed lines is constant - the graphics display always starts at the same point.

Then how do we deal with the location values that are not evenly dividable by eight ? Now, lets assume that the location is L, and we have C, which is the location divided by eight (C = L/8), and R, which is the remainder (R = L%8). To make the picture scroll to the right position, we need to delay the bad line less than before - R lines less for location L than for location C*8. E.g. for location

2 we delay the bad line two lines less than for location 0. This also shows that we need 7 lines more than is needed for to compensate for the linecrunch.

Determining the number of linecrunch lines is a recursive process, because when you use more linecrunch lines, that decreases the number of lines you have available for the display and you need bigger range for the location value. The linecrunch can be started after 12 lines, and we need at least 7 lines to use the soft y-scroll. This makes 181 lines available for the display originally.

Because we need to show 400 lines of graphics, we would need (400-181)/8=28 linecrunch lines. However, this in turn reduces the number of lines we have for graphics to 181-28=153 and we need (400-153)/8=31 linecrunch lines. Again, 181-31 is 150. We get (400-150)/8=32 and there it finally converges and we have 149 lines for graphics, which makes location values 0..251 valid.

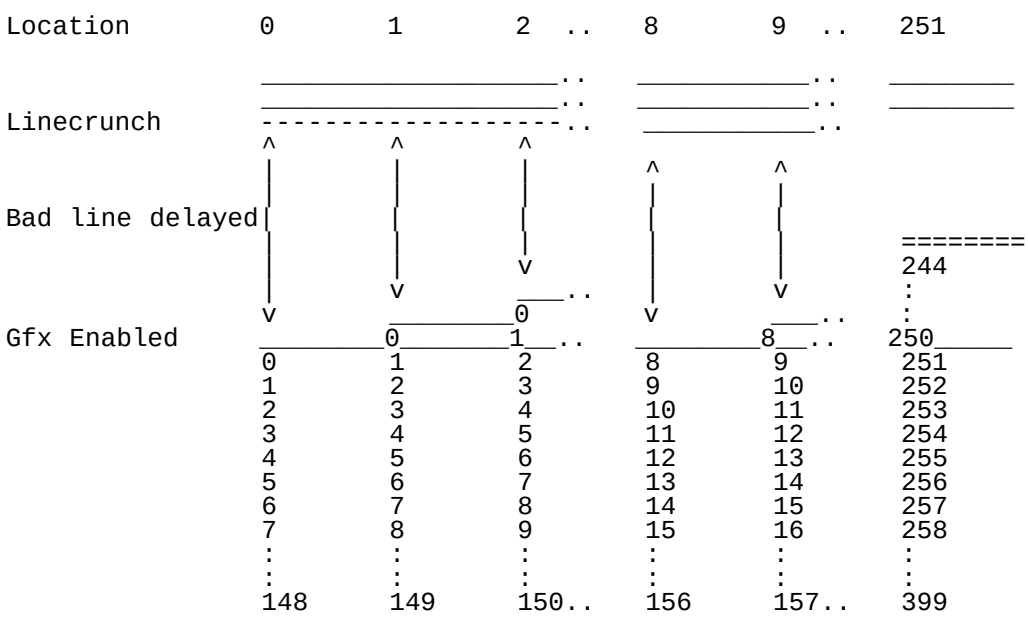


Figure 2: Linecrunch and DMA delay in BFLI
(Graphics lines not in scale)

Clipping added

Now we can scroll the picture to any location we want, but the top of the picture is not clipped and it is very annoying to watch. We need to enable the graphics at the same point regardless of the y-scroll value. The answer is in the extended color mode (ECM).

When both ECM and multicolor mode (MCM) are selected, VIC-II will turn the display to black. This is because there is a conflicting situation and it just can't decide which color scheme to use. The video accesses will continue to happen just like before, the data is just not displayed. When the ECM bit is cleared again, the normal multicolor graphics is shown.

So, we set the ECM bit and start to display the first eight lines of the FLI. Because the FLI routine already writes to $D011, we just make sure the ECM bit is set in the first R number of writes to $D011 and zero in all other.

The viewer is now 'complete'. You can take a look at the code below or you can get C64Gfx1_4.lha and see it in action yourself and not just rely on my word. The package includes converter programs for BFLI, FLI and Koala (ANSI-C), couple of example pictures and viewers for PAL and NTSC machines.

-Pasi 'Albert' Ojala albert@cs.tut.fi

BFLI viewer program for PAL machines

```
UPOS   = $C00   ; temporary area for tables
BANK   = $D00   ;  UPOS for linecrunch, BANK for FLI bank select
RASTER = 29     ; where to position the sprite -> IRQ 20 lines later
DUMMY  = $FFF   ; dummy location for timing purposes
FLISZ  = 19-1   ; visible FLI size in character rows - 1

*= $810
        SEI
        LDA #$7F:STA $DC0D      ; IRQ setup
        LDA #1:STA $D01A
        STA $D015:STA KEYW+1
        LDA #<IRQ:STA $314
        LDA #>IRQ:STA $315
        LDA #RASTER:STA $D001:CLC:ADC #20:STA $D012
        LDA #0:STA $D017
        LDA #0:STA 2
        JSR NEWPOS              ; Init the FLI routines
        LDA #$A:STA $D011       ; Blank screen
        LDX #23                 ; Init tables
BLOOP   LDA #$94:STA BANK,X
        LDA #$96:STA BANK+24,X
        DEX:BPL BLOOP
        LDX #15
LOOP0   LDA YINIT,X:AND #$77    ; Change to $37 to better see the
        STA UPOS,X              ;  workings of the routines
        STA UPOS+16,X
        STA UPOS+32,X
        DEX:BPL LOOP0

        LDA #$34:STA 1          ; Copy to the last video bank
        LDA #$80:STA SRC+2      ; from $8000-$BFFF to $C000-$FFFF
        LDA #$C0:STA DST+2
        LDX #0:LDY #$40
SRC     LDA $8000,X
DST     STA $C000,X
        INX:BNE SRC
        INC SRC+2:INC DST+2
        DEY:BNE SRC
        LDA #$37:STA 1

        LDX #0                  ; Init color memory
LP      LDA $3C00,X:STA $D800,X ; All 1024 bytes are used
        LDA $3D00,X:STA $D900,X ;  - some even twice!
        LDA $3E00,X:STA $DA00,X
        LDA $3F00,X:STA $DB00,X
        INX:BNE LP
        LDA $DC0D:CLI

KEYW    LDX #0:BNE KEYW         ; Wait for space to be pressed
        SEI                     ; System to normal
        LDA #$37:STA 1
        JSR $FDA3
        LDA #$97:STA $DD00
        JSR $E5A0
        LDY #3
IRQL    LDA $FD30,Y:STA $314,Y
        DEY:BPL IRQL

        LDX #0:LDA #1           ; Clear color memory
CLL     STA $D800,X:STA $D900,X
        STA $DA00,X:STA $DB00,X
        INX:BNE CLL
        CLI:RTS

YINIT   BYT $78,$79,$7A,$7B,$7C,$7D,$7E,$7F
        BYT $78,$79,$7A,$7B,$7C,$7D,$7E,$7F

*=*-<*+256

IRQ     LDA #$18:STA $D016:LDX #0:LDA #$5A
        INC DUMMY:DEC DUMMY             ; Synchronization
        STX $D020:STX $D021:STA $D011

        LDA #$15:STA $D018
        LDA #$97:STA $DD00
        LDX #44                 ; Wait for the 4th line
LL      DEX:BPL LL:NOP
        LDX #0

LOOP3   NOP                     ; Linecrunch-part routine
        LDA UPOS+6,X:INC DUMMY:STA $D011
```

```
        NOP:NOP:INC DUMMY
        NOP:NOP:NOP:NOP:NOP
        NOP:NOP:NOP:NOP:NOP
        NOP:NOP:NOP:NOP:NOP
        INX
E1      CPX #$10:BNE LOOP3      ; Skip that many character rows-4
        BIT $EA
LOOP4   LDA UPOS,X:INC DUMMY:STA $D011
        NOP:NOP:NOP:INC DUMMY
        NOP:NOP:NOP:NOP:NOP
        NOP:NOP:NOP:NOP:NOP
        NOP:NOP:NOP:NOP:LDA #0
        INX
E2      CPX #$1F:BNE LOOP4      ; Delay DMA until we are at the
                                ;  'same place' each time

        LDA #0:STA $D020        ; Now wait for the bad line and start FLI
        BIT $EA:NOP
        NOP:NOP:NOP:NOP
        NOP:NOP:NOP:NOP
        NOP:NOP:NOP:NOP
B0      LDA #$92:STA $DD00:NOP  ; The right video bank

        ; Wait for 0-7 lines to set the ECM mode off
        ;  (makes the graphics visible)

F0      LDA #0:STA $D011:LDA #$08:STA $D018:NOP:NOP:NOP:NOP:BIT $EA
F1      LDA #0:STA $D011:LDA #$18:STA $D018:NOP:NOP:NOP:NOP:BIT $EA
F2      LDA #0:STA $D011:LDA #$28:STA $D018:NOP:NOP:NOP:NOP:BIT $EA
F3      LDA #0:STA $D011:LDA #$38:STA $D018:NOP:NOP:NOP:NOP:BIT $EA
F4      LDA #0:STA $D011:LDA #$48:STA $D018:NOP:NOP:NOP:NOP:BIT $EA
F5      LDA #0:STA $D011:LDA #$58:STA $D018:NOP:NOP:NOP:NOP:BIT $EA
F6      LDA #0:STA $D011:LDA #$68:STA $D018:NOP:NOP:NOP:NOP:BIT $EA
F7      LDA #0:STA $D011:LDA #$78:STA $D018
        LDX #FLISZ:NOP:NOP:NOP:BIT $EA

        ; Do FLI 18 more character rows

F8      LDA #0:STA $D011:LDA #$08:STA $D018
B1      LDA BANK,X:STA $DD00:BIT $EA
F9      LDA #0:STA $D011:LDA #$18:STA $D018:NOP:NOP:NOP:NOP:BIT $EA
FA      LDA #0:STA $D011:LDA #$28:STA $D018:NOP:NOP:NOP:NOP:BIT $EA
FB      LDA #0:STA $D011:LDA #$38:STA $D018:NOP:NOP:NOP:NOP:BIT $EA
FC      LDA #0:STA $D011:LDA #$48:STA $D018:NOP:NOP:NOP:NOP:BIT $EA
FD      LDA #0:STA $D011:LDA #$58:STA $D018:NOP:NOP:NOP:NOP:BIT $EA
FE      LDA #0:STA $D011:LDA #$68:STA $D018:NOP:NOP:NOP:NOP:BIT $EA
FF      LDA #0:STA $D011:LDA #$78:STA $D018:NOP:NOP:DEX:BMI EFLI:JMP F8
EFLI    NOP
        LDA #$FC
WL      CMP $D012:BNE WL
        INC DUMMY:INC DUMMY:INC DUMMY:INC DUMMY
        INC DUMMY:INC DUMMY:INC DUMMY:INC DUMMY
        INC DUMMY:INC DUMMY:STA $D020

        JSR NEWPOS      ; Update the location
        JSR CHPOS       ; Change to a new location
        LDA $DC01:AND #$10:BNE OV3      ; Check for the space bar
        LDA #0:STA KEYW+1
OV3     LDX #$53:STX $D011:INC $D019:JMP $EA81

NEWPOS  LDA #0          ; Init the IRQ routine for this position
        LSR:LSR:CLC:ADC #4:STA E1+1
        LDA #7:SEC:SBC NEWPOS+1:AND #7:TAX:TAY:CLC:ADC #35:STA E2+1
        LDA UPOS+3+7,Y:DEX:BMI J0:AND #$3F
J0      STA F7+1:AND #$3F:STA FF+1
        LDA UPOS+3+6,Y:DEX:BMI J1:AND #$3F
J1      STA F6+1:AND #$3F:STA FE+1
        LDA UPOS+3+5,Y:DEX:BMI J2:AND #$3F
J2      STA F5+1:AND #$3F:STA FD+1
        LDA UPOS+3+4,Y:DEX:BMI J3:AND #$3F
J3      STA F4+1:AND #$3F:STA FC+1
        LDA UPOS+3+3,Y:DEX:BMI J4:AND #$3F
J4      STA F3+1:AND #$3F:STA FB+1
        LDA UPOS+3+2,Y:DEX:BMI J5:AND #$3F
J5      STA F2+1:AND #$3F:STA FA+1
        LDA UPOS+3+1,Y:DEX:BMI J6:AND #$3F
J6      STA F1+1:AND #$3F:STA F9+1
        LDA UPOS+3+0,Y:DEX:BMI J7:AND #$3F
J7      STA F0+1:AND #$3F:STA F8+1
        LDA #$96:STA B0+1:LDA #199:SEC:SBC NEWPOS+1:BCC OV2
        LSR:LSR:LSR:CLC:ADC #5:STA B1+1
```

```
        RTS
OV2     LDA #0:STA B1+1:LDX #$94:STX B0+1:RTS

CHPOS   LDX NEWPOS+1
        LDA $DC00:TAY           ; Get joystick
        AND #$10:BNE DIR        ; If no button pressed
        TYA:AND #1:BEQ UP       ; If joy up
        TYA:AND #2:BEQ DOWN     ; If joy down
        RTS
DIR     LDA #0:BEQ UP
DOWN    DEX:CPX #$FF:BNE DOK
        LDX #0:STX DIR+1        ; Change direction
DOK     STX NEWPOS+1:RTS
UP      INX:CPX #$FD:BCC UOK    ; 251(locations)+149(visible)=400
        LDX #$FC:STX DIR+1      ; Change direction
UOK     STX NEWPOS+1:RTS
```

The BFLI file format:

	Lines	File Offset	BFLI Display Offset	Lines	Size
Colors	0-1.3	0..55	944..999	22.7-24	56
I	1.3-2	56..79	-		24
	2-24	80..999	0..919	0-22	920
	24-24.7	1000..1023	920..943	22-22.7	24
II	0-1.3	0..55	1968..2024	49.3-50.6	56
	1.3-24.7	56..1023	1000..1967	24-49.3	968
Gfx	0-1.3	0..447	7552..7999	22.7-24	448
I	1.3-2	448..639	-		192
	2-24	640..7999	0..7359	0-22	7360
	24-24.7	8000..8191	7360..7551	22-22.7	192
II	0-1.3	0..447	15744..16192	49.3-50.6	448
	1.3-24.7	448..8191	8000..15743	24-49.3	7744

==========

Making stable raster routines (C64 and VIC-20)
by Marko Makela (Marko.Makela@HUT.FI)

Preface

Too many graphical effects, also called raster effects, have been coded in a very sloppy way. For instance, if there are any color bars on the screen in a game or demo, the colors often jitter a bit, e.g. they are not stable. And also, it is far too easy to make virtually any demo crash by hitting the Restore key, or at least cause visual distortions on the screen.

As late as a year ago I still hadn't coded a stable raster interrupt routine myself. But then I had to do it, since I was researching the video chip timing details together with my German friend Andreas Boose. It was ashaming that we had the same level of knowledge when it came to the hardware, but he was the only of us who had written a stable raster routine. Well, finally I made me to start coding. I used the same double-interrupt idea as Andreas used in his routine.

After a couple of errors my routine worked, and I understood how it works exactly. (This is something that separates us normal coders from demo people: They often code by instinct; by patching the routine until it works, without knowing exactly what is happening. That's why demos often rely on weird things, like crash if the memory is not initialized properly.)

In this article, I document two methods of creating stable raster routines on Commodore computers. The principles apply for most 8-bit computers, not only Commodores, but raster effects are very rarely seen on other computers.

Background

What are raster effects? They are effects, where you change the screen appearance while it is being drawn. For instance, you can set the screen color to white in the top of the screen, and to black in the middle of the screen. In that way, you will get a picture whose

top half is white and bottom half black. Normally such effects are implemented with interrupt routines that are executed synchronized with the screen refresh.

The video chip on the Commodore 64 and many other videochips have a special interrupt feature called the Raster interrupt. It will generate an IRQ in the beginning of a specified raster line. On other computers, like the VIC-20, there is no Raster interrupt, but you can generate the interrupts with a timer, provided that the timer and the videochip are clocked from the same source.

Even if the processor gets an interrupt signal at the same position on each video frame, it won't always be executing the first instruction of the interrupt routine at the same screen position. The NMOS 6502 machine instructions can take 2 to 9 machine cycles to execute, and if the main program contains instructions of very varying lengths, the beginning position of the interrupt can jump between 7 different positions. This is why you need to synchronize the raster routine when doing serious effects.

Also, executing the interrupt sequence will take 7 additional cycles, and the interrupt sequence will only start after the current instruction if the interrupt arrived at least two cycles before the end of the current instruction. It is even possible that an interrupt arrives while interrupts are disabled and the processor is just starting to execute a CLI instruction. Alas, the processor will not jump to the interrupt right after the CLI, but it will execute the next instruction before jumping to it. This is natural, since the CLI takes only two cycles. But anyway, this is only a constant in our equation, and actually out of the scope of this article.

How to synchronize a raster interrupt routine? The only way is to check the current screen position and delay appropriately many cycles. There are several ways of doing this, some of which are very awful and inefficient. The ugliest ways of doing this on the Commodore 64 I know are busy-waiting several raster lines and polling the raster line value, or using the Light pen feature, which will fail if the user presses the fire button on Joystick port 1. Here I will present two ways, both very elegant in my opinion.

Using an auxiliary timer

On the VIC-20, there is no Raster interrupt feature in the video chip. All you can do is to use a timer for generating raster interrupts. And if you use two timers running at a constant phase difference, you can get full synchronization. The first timer generates the raster interrupt, and the second timer, the auxiliary timer, tells the raster routine where it is running. Actually you could even use the first timer also for the checking, but the code will look nicer in the way I will be presenting now. Besides, you can use the auxiliary timer idea even when real raster interrupts are available.

The major drawback of using an auxiliary timer is initializing it. The initialization routine must synchronize with the screen, that is, wait for the beginning of the wanted raster line. To accomplish this, the routine must first wait for a raster line that occurs a bit earlier. About the only way to do this is with a loop like

```
        LDA #value
loop    CMP raster
        BNE loop
```

One round of this loop will take 4+3=7 cycles to execute, assuming that absolute addressing is being used. The loop will be finished if the raster register contains the wanted value while the processor reads it on the last cycle of the CMP instruction. The raster register can actually have changed already on the first cycle of the BNE instruction on the previous run of the loop, that is 7 cycles earlier!

Because of this, the routine must poll the raster register for several raster lines, always consuming one cycle more if the raster register changed too early. As the synchronization can be off at most by 7 cycles, a loop of 7 raster register value changes would do, but I made the loop a bit longer in my VIC-20 routine. (Well, I have to admit it, I was too lazy to make it work only with 7 rounds.)

After the initialization routine is fully synchronized the screen, it can set up the timer(s) and interrupts and exit. The auxiliary timer in my VIC-20 demo routine is several dozens of cycles after the

primary timer, see the source code for comments. It is arranged so that the auxiliary timer will be at least 0 when it is being read in the raster routine. The raster routine will wait as many extra cycles as the auxiliary timer reads, however at most 15 cycles.

Using double raster interrupt

On the Commodore 64, I have never seen the auxiliary timer scheme being used. Actually I haven't seen it being used anywhere, I was probably the first one who made a stable raster interrupt routine on the VIC-20. Instead, the double interrupt method is becoming the standard on the C64 side.

The double interrupt method is based entirely on the Raster interrupt feature of the video chip. In the first raster interrupt routine, the program sets up another raster interrupt on a further line, changes the interrupt vector and enables interrupts.

In the place where the second raster interrupt will occur, there will be 2-byte instructions in the first interrupt routine. In this way, the beginning of the next raster interrupt will be off at most by one cycle. Some coders might not care about this one cycle, but if you can do it right, why wouldn't you do it right until the end?

At the beginning of the second raster interrupt routine, you will read the raster line counter register at the point where it is about to change. When the raster routine is being executed, there are two possibilities: Either the raster counter has just changed, or it will change on the next cycle. So, you just need to compare if the register changed one cycle too early or not, and delay a cycle when needed. This is easily accomplished with a branch to the next address.

Of course, somewhere in your second raster interrupt routine you must restore the original raster interrupt position and set the interrupt vector to point to the first interrupt routine.

Applying in practice

I almost forgot my complaints about demos crashing when you actively hit the Restore key. On the VIC-20, you can disable NMI interrupts generated by the Restore key, and on the C64, you can generate an NMI interrupt with the CIA2 timer and leave the NMI-line low, so that no further high-to-low transitions will be recognized on the line. The example programs demonstrate how to do this.

So far, this article has been pretty theoretical. To apply these results in practice, you must definitely know how many CPU clock cycles the video chip consumes while drawing a scan line. This is fairly easy to measure with a timer interrupt, if you patch the interrupt handler so that it changes the screen color on each run. Set the timer interval to LINES*COLUMNS cycles, where LINES is the amount of raster lines and COLUMNS is your guess for the amount of clock cycles spent in a raster line.

If your guess is right, the color will always be changed in the same screen position (neglecting the 7-cycle jitter). When adjusting the timer, remember that the timers on the 6522 VIA require 2 cycles for re-loading, and the ones on the 6526 CIA need one extra cycle. Keep trying different timer values until you the screen color changes at one fixed position.

Commodore used several different values for LINES and COLUMNS on its videochips. They never managed to make the screen refresh rate exactly 50 or 60 Hertz, but they didn't hesitate to claim that their computers comply with the PAL-B or NTSC-M standards. In the following tables I have gathered some information of some Commodore video chips.

NTSC-M systems:

Host	Chip ID	Crystal freq/Hz	Dot clock/Hz	Processor clock/Hz	Cycles/ line	Lines/ frame
VIC-20	6560-101	14318181	4090909	1022727	65	261
C64	6567R56A	14318181	8181818	1022727	64	262
C64	6567R8	14318181	8181818	1022727	65	263

Later NTSC-M video chips were most probably like the 6567R8. Note that the processor clock is a 14th of the crystal frequency on all

NTSC-M systems.

PAL-B systems:

Host	Chip ID	Crystal freq/Hz	Dot clock/Hz	Processor clock/Hz	Cycles/ line	Lines/ frame
VIC-20	6561-101	4433618	4433618	1108405	71	312
C64	6569	17734472	7881988	985248	63	312

On the PAL-B VIC-20, the crystal frequency is simultaneously the dot clock, which is BTW a 4th of the crystal frequency used on the C64. On the C64, the crystal frequency is divided by 18 to generate the processor clock, which in turn is multiplied by 8 to generate the dot clock.

The basic timings are the same on all 6569 revisions, and also on any later C64 and C128 video chips. If I remember correctly, these values were the same on the C16 videochip TED as well.

Note that the dot clock is 4 times the processor clock on the VIC-20, and 8 times that on the C64. That is, one processor cycle is half a character wide on the VIC-20, and a full character on a C64. I don't have exact measurements of the VIC-20 timing, but it seems that while the VIC-20 videochips draw the characters on the screen, it first reads the character code, and then, on the following video cycle, the appearance on the current character line. There are no bad lines, like on the C64, where the character codes (and colors) are fetched on every 8th raster line.

Those ones who got upset when I said that Commodore has never managed to make a fully PAL-B or NTSC-M compliant 8-bit computer should take a closer look at the "Lines/frame" columns. If that does not convince you, calculate the raster line rate and the screen refresh rate from the values in the table and see that they don't comply with the standards. To calculate the line rate, divide the processor clock frequency by the amount of cycles per line. To get the screen refresh rate, divide that frequency by the amount of raster lines.

The Code

OK, enough theory and background. Here are the two example programs, one for the VIC-20 and one for the C64. In order to fully understand them, you need to know the exact execution times of NMOS 6502 instructions. (All 8-bit Commodore computers use the NMOS 6502 processor core, except the C65 prototype, which used a inferior CMOS version with all nice poorly-documented features removed.) You should check the 64doc document, available on my WWW pages at http://www.hut.fi/~msmakela/cbm/emul/x64/64doc.html, or via FTP at ftp.funet.fi:/pub/cbm/documents/64doc. I can also e-mail it to you on request.

Also, I have written a complete description of the video timing on the 6567R56A, 6567R8 and 6569 video chips, which could maybe be turned into another C=Hacking article. The document is currently partially in English and partially in German. The English part is available from ftp.funet.fi as /pub/cbm/documents/pal.timing, and I can send copies of the German part (screen resolution, sprite disturbance measurements, and more precise timing information) via e-mail.

The code is written for the DASM assembler, or more precisely for a extended ANSI C port of it made by Olaf Seibert. This excellent cross-assembler is available at ftp.funet.fi in /pub/cbm/programming.

First the raster demo for the VIC-20. Note that on the VIC-20, the $9004 register contains the upper 8 bits of the raster counter. So, this register changes only on every second line. I have tested the program on my 6561-101-based VIC-20, but not on an NTSC-M system.

It was hard to get in contact with NTSC-M VIC-20 owners. Daniel Dallmann, who has a NTSC-M VIC-20, although he lives in Germany, ran my test to determine the amount of cycles per line and lines per frame on the 6560-101. Unfortunately, the second VIA of his VIC-20 is partially broken, and because of this, this program did not work on his computer. Craig Bruce ran the program once, and he reported that it almost worked. I corrected a little bug in the code, so that now the display should be stable on an NTSC-M system, too. But the actual raster effect, six 16*16-pixel boxes centered at the top border, are very likely to be off their position.

```
  processor 6502

NTSC    = 1
PAL     = 2

;SYSTEM = NTSC  ; 6560-101: 65 cycles per raster line, 261 lines
SYSTEM  = PAL   ; 6561-101: 71 cycles per raster line, 312 lines

#if SYSTEM & PAL
LINES = 312
CYCLES_PER_LINE = 71
#endif
#if SYSTEM & NTSC
LINES = 261
CYCLES_PER_LINE = 65
#endif
TIMER_VALUE = LINES * CYCLES_PER_LINE - 2

  .org $1001    ; for the unexpanded Vic-20

; The BASIC line

basic:
  .word 0$      ; link to next line
  .word 1995    ; line number
  .byte $9E     ; SYS token

; SYS digits

  .if (* + 8) / 10000
  .byte $30 + (* + 8) / 10000
  .endif
  .if (* + 7) / 1000
  .byte $30 + (* + 7) % 10000 / 1000
  .endif
  .if (* + 6) / 100
  .byte $30 + (* + 6) % 1000 / 100
  .endif
  .if (* + 5) / 10
  .byte $30 + (* + 5) % 100 / 10
  .endif
  .byte $30 + (* + 4) % 10
0$:
  .byte 0,0,0   ; end of BASIC program

start:
  lda #$7f
  sta $912e     ; disable and acknowledge interrupts
  sta $912d
  sta $911e     ; disable NMIs (Restore key)

;synchronize with the screen
sync:
  ldx #28       ; wait for this raster line (times 2)
0$:
  cpx $9004
  bne 0$        ; at this stage, the inaccuracy is 7 clock cycles
                ; the processor is in this place 2 to 9 cycles
                ; after $9004 has changed
  ldy #9
  bit $24
1$:
  ldx $9004
  txa
  bit $24
#if SYSTEM & PAL
  ldx #24
#endif
#if SYSTEM & NTSC
  bit $24
  ldx #21
#endif
  dex
  bne *-1       ; first spend some time (so that the whole
  cmp $9004     ; loop will be 2 raster lines)
  bcs *+2       ; save one cycle if $9004 changed too late
  dey
  bne 1$
                ; now it is fully synchronized
                ; 6 cycles have passed since last $9004 change
```

```
                ; and we are on line 2(28+9)=74

;initialize the timers
timers:
  lda #$40      ; enable Timer A free run of both VIAs
  sta $911b
  sta $912b

  lda #<TIMER_VALUE
  ldx #>TIMER_VALUE
  sta $9116     ; load the timer low byte latches
  sta $9126

#if SYSTEM & PAL
  ldy #7        ; make a little delay to get the raster effect to the
  dey           ; right place
  bne *-1
  nop
  nop
#endif
#if SYSTEM & NTSC
  ldy #6
  dey
  bne *-1
  bit $24
#endif

  stx $9125     ; start the IRQ timer A
                ; 6560-101: 65 cycles from $9004 change
                ; 6561-101: 77 cycles from $9004 change
  ldy #10       ; spend some time (1+5*9+4=55 cycles)
  dey           ; before starting the reference timer
  bne *-1
  stx $9115     ; start the reference timer

pointers:
  lda #<irq     ; set the raster IRQ routine pointer
  sta $314
  lda #>irq
  sta $315
  lda #$c0
  sta $912e     ; enable Timer A underflow interrupts
  rts           ; return

irq:
; irq (event)   ; > 7 + at least 2 cycles of last instruction (9 to 16 total)
; pha           ; 3
; txa           ; 2
; pha           ; 3
; tya           ; 2
; pha           ; 3
; tsx           ; 2
; lda $0104,x   ; 4
; and #xx       ; 2
; beq           ; 3
; jmp ($314)    ; 5
                ; ---
                ; 38 to 45 cycles delay at this stage

  lda $9114     ; get the NMI timer A value
                ; (42 to 49 cycles delay at this stage)
; sta $1e00     ; uncomment these if you want to monitor
; ldy $9115     ; the reference timer on the screen
; sty $1e01
  cmp #8        ; are we more than 7 cycles ahead of time?
  bcc 0$
  pha           ; yes, spend 8 extra cycles
  pla
  and #7        ; and reset the high bit
0$:
  cmp #4
  bcc 1$
  bit $24       ; waste 4 cycles
  and #3
1$:
  cmp #2        ; spend the rest of the cycles
  bcs *+2
  bcs *+2
  lsr
  bcs *+2       ; now it has taken 82 cycles from the beginning of the IRQ
```

```
effect:
  ldy #16       ; perform amazing video effect
  lda $900f
  tax
  eor #$f7
0$:
  sta $900f
  stx $900f
  sta $900f
  stx $900f
  sta $900f
  stx $900f
  sta $900f
  stx $900f
  sta $900f
  stx $900f
  sta $900f
  stx $900f
  pha
  pla
#if SYSTEM & PAL
  pha
  pla
  nop
#endif
#if SYSTEM & NTSC
  bit $24
#endif
  nop
  dey
  bne 0$        ; end of amazing video effect

  jmp $eabf     ; return to normal IRQ
```

And after you have recovered from the schock of seeing a VIC-20 program, here is an example for the C64. It does also something noteworthy; it removes the side borders on a normal screen while displaying all eight sprites. Well, it cannot remove the borders on bad lines, and the bad lines look pretty bad. But I could use the program for what I wanted: I measured the sprite distortions on all videochip types I had at hand. (FYI: the sprites 0-2 get distorted at the very right of the screen, and the sprites 6 and 7 are invisible at the very left of the screen. You will need a monitor with horizontal size controls to witness these effects.)

This program is really robust, it installs itself nicely to the interrupt routine chain. It even has an entry point for deinstalling itself. But in its robustness it uses self-modifying code to store the original interrupt routine address. :-)

The code also relies on the page boundaries in being where they are. The cycles are counted so that the branches "irqloop" must take 4 cycles. If the "irqloop" comes to the same CPU page with the branch instructions, you must add one cycle to the loop in a way or another. When coding the routine, I noticed again how stupid assembly coding can be, especially conditional assembling. In a machine language monitor you have far better control on page boundaries. BTW, you might wonder why I disable the Restore key in a subroutine at the end and not in the beginning of the program. Well, the routine was so long that it would have affected the "irqloop" page boundaries. And I didn't want to risk the modified programs working on all three different videochip types on the first try.

In the code, there are some comments that document the video timing, like this one:

```
;3s4s5s6s7srrrrrgggggggggggggggggggggggggggggggggggggggg--||0s1s2s Phi-1 VIC-II
;ssssssssss                                               ||ssssss Phi-2 VIC-II
;==========xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx||XXX====== Phi-2 6510
;          ^ now we are here
```

The two vertical bars "|" denote optional cycles. On PAL-B systems (63 cycles per line), they are not present. On 6567R56A, which has 64 cycles per line, there is one additional cycle on this position, and the 6567R8 has two additional cycles there.

The numbers 0 through 7 are sprite pointer fetches (from the end of the character matrix, e.g. the text screen), the "s" characters denote sprite image fetches, the "r"s are memory refresh, and the "g" are

```
graphics fetches.  The two idle video chip cycles are marked with "-".
On the processor timing line, the "=" signs show halted CPU, "x" means
free bus, and "X" means that the processor will be halted at once,
unless it is performing write cycles.

  processor 6502

; Select the video timing (processor clock cycles per raster line)
CYCLES = 65     ; 6567R8 and above, NTSC-M
;CYCLES = 64    ; 6567R5 6A, NTSC-M
;CYCLES = 63    ; 6569 (all revisions), PAL-B

cinv = $314
cnmi = $318
raster = 52     ; start of raster interrupt
m = $fb         ; zero page variable

  .org $801
basic:
  .word 0$      ; link to next line
  .word 1995    ; line number
  .byte $9E     ; SYS token

; SYS digits

  .if (* + 8) / 10000
  .byte $30 + (* + 8) / 10000
  .endif
  .if (* + 7) / 1000
  .byte $30 + (* + 7) % 10000 / 1000
  .endif
  .if (* + 6) / 100
  .byte $30 + (* + 6) % 1000 / 100
  .endif
  .if (* + 5) / 10
  .byte $30 + (* + 5) % 100 / 10
  .endif
  .byte $30 + (* + 4) % 10

0$:
  .byte 0,0,0   ; end of BASIC program

start:
  jmp install
  jmp deinstall

install:        ; install the raster routine
  jsr restore   ; Disable the Restore key (disable NMI interrupts)
checkirq:
  lda cinv      ; check the original IRQ vector
  ldx cinv+1    ; (to avoid multiple installation)
  cmp #<irq1
  bne irqinit
  cpx #>irq1
  beq skipinit
irqinit:
  sei
  sta oldirq    ; store the old IRQ vector
  stx oldirq+1
  lda #<irq1
  ldx #>irq1
  sta cinv      ; set the new interrupt vector
  stx cinv+1
skipinit:
  lda #$1b
  sta $d011     ; set the raster interrupt location
  lda #raster
  sta $d012
  ldx #$e
  clc
  adc #3
  tay
  lda #0
  sta m
0$:
  lda m
  sta $d000,x   ; set the sprite X
  adc #24
  sta m
  tya
  sta $d001,x   ; and Y coordinates
```

```
  dex
  dex
  bpl 0$
  lda #$7f
  sta $dc0d     ; disable timer interrupts
  sta $dd0d
  ldx #1
  stx $d01a     ; enable raster interrupt
  lda $dc0d     ; acknowledge CIA interrupts
  lsr $d019     ; and video interrupts
  ldy #$ff
  sty $d015     ; turn on all sprites
  cli
  rts

deinstall:
  sei           ; disable interrupts
  lda #$1b
  sta $d011     ; restore text screen mode
  lda #$81
  sta $dc0d     ; enable Timer A interrupts on CIA 1
  lda #0
  sta $d01a     ; disable video interrupts
  lda oldirq
  sta cinv      ; restore old IRQ vector
  lda oldirq+1
  sta cinv+1
  bit $dd0d     ; re-enable NMI interrupts
  cli
  rts

; Auxiliary raster interrupt (for syncronization)
irq1:
; irq (event)   ; > 7 + at least 2 cycles of last instruction (9 to 16 total)
; pha           ; 3
; txa           ; 2
; pha           ; 3
; tya           ; 2
; pha           ; 3
; tsx           ; 2
; lda $0104,x   ; 4
; and #xx       ; 2
; beq           ; 3
; jmp ($314)    ; 5
                ; ---
                ; 38 to 45 cycles delay at this stage
  lda #<irq2
  sta cinv
  lda #>irq2
  sta cinv+1
  nop           ; waste at least 12 cycles
  nop           ; (up to 64 cycles delay allowed here)
  nop
  nop
  nop
  nop
  inc $d012     ; At this stage, $d012 has already been incremented by one.
  lda #1
  sta $d019     ; acknowledge the first raster interrupt
  cli           ; enable interrupts (the second interrupt can now occur)
  ldy #9
  dey
  bne *-1       ; delay
  nop           ; The second interrupt will occur while executing these
  nop           ; two-cycle instructions.
  nop
  nop
  nop
oldirq = * + 1  ; Placeholder for self-modifying code
  jmp *         ; Return to the original interrupt

; Main raster interrupt
irq2:
; irq (event)   ; 7 + 2 or 3 cycles of last instruction (9 or 10 total)
; pha           ; 3
; txa           ; 2
; pha           ; 3
; tya           ; 2
; pha           ; 3
; tsx           ; 2
; lda $0104,x   ; 4
```

```
; and #xx       ; 2
; beq           ; 3
; jmp (cinv)    ; 5
                ; ---
                ; 38 or 39 cycles delay at this stage
  lda #<irq1
  sta cinv
  lda #>irq1
  sta cinv+1
  ldx $d012
  nop
#if CYCLES - 63
#if CYCLES - 64
  nop           ; 6567R8, 65 cycles/line
  bit $24
#else
  nop           ; 6567R56A, 64 cycles/line
  nop
#endif
#else
  bit $24       ; 6569, 63 cycles/line
#endif
  cpx $d012     ; The comparison cycle is executed CYCLES or CYCLES+1 cycles
                ; after the interrupt has occurred.
  beq *+2       ; Delay by one cycle if $d012 hadn't changed.
                ; Now exactly CYCLES+3 cycles have passed since the interrupt.
  dex
  dex
  stx $d012     ; restore original raster interrupt position
  ldx #1
  stx $d019     ; acknowledge the raster interrupt
  ldx #2
  dex
  bne *-1
  nop
  nop
  lda #20       ; set the amount of raster lines-1 for the loop
  sta m
  ldx #$c8
irqloop:
  ldy #2
  dey
  bne *-1       ; delay
  dec $d016     ; narrow the screen (exact timing required)
;3s4s5s6s7srrrrrgggggggggggggggggggggggggggggggggggggggg--||0s1s2s Phi-1 VIC-II
;sssssssss                                                 ||ssssss Phi-2 VIC-II
;==========xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx||XXX====== Phi-2 6510
;          ^ now we are here
  stx $d016     ; expand the screen
#if CYCLES - 63
#if CYCLES - 64
  bit $24       ; 6567R8
#else
  nop           ; 6567R56A
#endif
#else
  nop           ; 6569
#endif
  dec m
  bmi endirq
  clc
  lda $d011
  sbc $d012
  and #7
  bne irqloop   ; This instruction takes 4 cycles instead of 3,
                ; because the page boundary is crossed.
badline:
  dec m
  nop
  nop
  nop
  nop
  dec $d016
;3s4s5s6s7srrrrrgggggggggggggggggggggggggggggggggggggggg--||0s1s2s Phi-1 VIC-II
;sssssssss    ccccccccccccccccccccccccccccccccccccccccc    ||ssssss Phi-2 VIC-II
;==========xXXX=========================================||***====== Phi-2 6510
;          ^ we are here
  stx $d016
;3s4s5s6s7srrrrrgggggggggggggggggggggggggggggggggggggggg--||0s1s2s Phi-1 VIC-II
;sssssssss                                                 ||ssssss Phi-2 VIC-II
;==========xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx||XXX====== Phi-2 6510
```

```
;          ^ ^^- we are here (6569)
;          | \- or here (6567R56A)
;          \- or here (6567R8)
  ldy #2
  dey
  bne *-1
  nop
  nop
#if CYCLES - 63
#if CYCLES - 64
  nop            ; 6567R8, 65 cycles/line
  nop
  nop
#else
  bit $24        ; 6567R56A, 64 cycles/line
#endif
#else
  nop            ; 6569, 63 cycles/line
#endif
  dec m
  bpl irqloop    ; This is a 4-cycle branch (page boundary crossed)
endirq:
  jmp $ea81      ; return to the auxiliary raster interrupt

restore:         ; disable the Restore key
  lda cnmi
  ldy cnmi+1
  pha
  lda #<nmi      ; Set the NMI vector
  sta cnmi
  lda #>nmi
  sta cnmi+1
  ldx #$81
  stx $dd0d      ; Enable CIA 2 Timer A interrupt
  ldx #0
  stx $dd05
  inx
  stx $dd04      ; Prepare Timer A to count from 1 to 0.
  ldx #$dd
  stx $dd0e      ; Cause an interrupt.
nmi = * + 1
  lda #$40       ; RTI placeholder
  pla
  sta cnmi
  sty cnmi+1     ; restore original NMI vector (although it won't be used)
  rts
```

Binaries

Here are the programs in uuencoded format. First the VIC-20 programs:

Color boxes for the VIC-20, NTSC-M version (probably distorted display):

```
begin 644 copper.6560
M`1``*$,L'GC0Q,#D```"I?XTND8TMD8T>D:(<[`20T/N@"20DK@20BB0D)"2B
M%<K0_<T$D+``B-#KJ4"-&Y&-*Y&I0Z)"C1:1C2:1H`:(T/TD)(XED:`*B-#]
MCA61J6R-%`.I$(T5`ZG`C2Z18*T4D<D(D`1(:"D'R020!"0D*0/)`K``L`!*
ML`"@$*T/D*I)]XT/D(X/D(T/D(X/D(T/D(X/D(T/D(X/D(T/D(X/D(T/D(X/
`D$AH)"3JB-#43+_J
`
end
```

Color boxes for the VIC-20, PAL-B version:

```
begin 644 copper.6561
M`1``*$,L'GC0Q,#D```"I?XTND8TMD8T>D:(<[`20T/N@"20DK@20BB0DHAC*
MT/W-!)"P`(C0[:E`C1N1C2N1J8:B5HT6D8TFD:`'B-#]ZNJ.)9&@"HC0_8X5
MD:EJC10#J1"-%0.IP(TND6"M%)')")`$2&@I!\D$D`0D)"D#R0*P`+``2K``
MH!"M#Y"J2?>-#Y".#Y"-#Y".#Y"-#Y".#Y"-#Y".#Y"-#Y".#Y"-#Y".#Y!(
+:$AHZNJ(T--,O^J.
`
end
```

Removed sideborders with 8 sprites and bad lines, PAL-B version:

```
begin 644 raster.63
M`0@*",L'GC(P-C$```!,$PA,=`@@'0FM%`.N%0/)E=`$X`CP$7B-N0B.N@BI
ME:((C10#CA4#J1N-$="I-(T2T*(.&&D#J*D`A?NE^YT`T&D8A?N8G0'0RLH0
```

```
M[ZE_C0W<C0W=H@&.&M"M#=Q.&="@_XP5T%A@>*D;C1'0J8&-#=RI`(T:T*VY
M"(T4`ZVZ"(T5`RP-W5A@J;N-%`.I"(T5`^KJZNKJZNX2T*D!C1G06*`)B-#]
MZNKJZNI,N`BIE8T4`ZD(C14#KA+0ZB0D[!+0\`#*RHX2T*(!CAG0H@+*T/WJ
MZJD4A?NBR*`"B-#]SA;0CA;0ZL;[,",8K1'0[1+0*0?0Y<;[ZNKJZLX6T(X6
MT*`"B-#]ZNKJQOL0S4R!ZJT8`ZP9`TBI0HT8`ZD)C1D#HH&.#=VB`(X%W>B.
1!-VBW8X.W:E`:(T8`XP9`V`8
`
end


Removed sideborders with 8 sprites and bad lines, 6567R56A version
(very old NTSC-M C64s):

begin 644 raster.64
M`0@*",L'GC(P-C$```!,$PA,=`@@'@FM%`.N%0/)E=`$X`CP$7B-N0B.N@BI
ME:((C10#CA4#J1N-$="I-(T2T*(.&&D#J*D`A?NE^YT`T&D8A?N8G0'0RLH0
M[ZE_C0W<C0W=H@&.&M"M#=Q.&="@_XP5T%A@>*D;C1'0J8&-#=RI`(T:T*VY
M"(T4`ZVZ"(T5`RP-W5A@J;N-%`.I"(T5`^KJZNKJZNX2T*D!C1G06*`)B-#]
MZNKJZNI,N`BIE8T4`ZD(C14#KA+0ZNKJ[!+0\`#*RHX2T*(!CAG0H@+*T/WJ
MZJD4A?NBR*`"B-#]SA;0CA;0ZL;[,"08K1'0[1+0*0?0Y<;[ZNKJZLX6T(X6
MT*`"B-#]ZNHD),;[$,Q,@>JM&`.L&0-(J4.-&`.I"8T9`Z*!C@W=H@".!=WH
2C@3=HMV.#MVI0&B-&`.,&0-@
`
end

Removed sideborders with 8 sprites and bad lines, 6567R8 and above
(not too old NTSC-M C64s and all C128s)

begin 644 raster.65
M`0@*",L'GC(P-C$```!,$PA,=`@@(0FM%`.N%0/)E=`$X`CP$7B-N0B.N@BI
ME:((C10#CA4#J1N-$="I-(T2T*(.&&D#J*D`A?NE^YT`T&D8A?N8G0'0RLH0
M[ZE_C0W<C0W=H@&.&M"M#=Q.&="@_XP5T%A@>*D;C1'0J8&-#=RI`(T:T*VY
M"(T4`ZVZ"(T5`RP-W5A@J;N-%`.I"(T5`^KJZNKJZNX2T*D!C1G06*`)B-#]
MZNKJZNI,N`BIE8T4`ZD(C14#KA+0ZNHD).P2T/``RLJ.$M"B`8X9T*("RM#]
MZNJI%(7[HLB@`HC0_<X6T(X6T"0DQOLP)1BM$=#M$M`I!]#DQOOJZNKJSA;0
MCA;0H`*(T/WJZNKJZL;[$,I,@>JM&`.L&0-(J4:-&`.I"8T9`Z*!C@W=H@".
5!=WHC@3=HMV.#MVI0&B-&`.,&0-@
`
end


That was all, folks!  I hope you learned something from this article.
Feel free to e-mail me at Marko.Makela@HUT.FI, should anything remain
unclear.
```

```
==========================================================================
```

A Different Perspective, part III
by Stephen Judd --- sjudd@nwu.edu
   George Taylor --- aa601@cfn.cs.dal.ca

Whew! What a busy time it's been -- research to get done, conferences, classes... between getting things done and blowing other things off, I one day reflected for a moment and realized that I had three days left to get the next article together for C=Hacking! So everything has been slapped together at the last minute, and I hope you'll forgive any bugs or unclear concepts.

>>> ANECDOTE ALERT <<<

And that reminds me: I just got JiffyDOS and an FD-2000 drive -- what a wonderful device. I have a 1.6 megabyte disk formatted into three partitions. The first contains my Merlin 128 assembler, the second is some 4000 blocks large and I use it for all my various versions of code while debugging, and the third is maybe 1000 blocks, and contains only finished code -- no more swapping disks, no more deleting old versions that I hope I don't need to make room on the disk. Also, when I installed JiffyDOS I found a serious bug in my 128D -- a cricket, dead among the IC's.

This time we will cover a lot of ground which isn't so much cutting-edge as it is very useful. Let's face it: cubes are getting more than a little dull. A worthy end goal is to have a completely general routine for plotting a series of polygons -- that is, you supply a list of (x,y,z) coordinates from which the program can form a list of polygons. These polygons may then be displayed in 2D, rotated, magnified, filled, etc. And, much to my three-day astonishment, that is exactly what we are going to do.

But first, a little excursion. One thing we are of course always thinking about is optimization possibilities: in the shower, while sleeping/dreaming, out on dates, etc. So, where to begin? The biggest cycle hogs in the program are line drawing and face filling -- well, filling faces is pretty straightforward. What about line drawing?

Well, one downer of the routine is that every single pixel is

plotted. But as we know, on a computer any given line is made up of several smaller vertical and horizontal lines -- wouldn't it be neat if we could think of a way to plot these line chunks all at once, instead of a pixel at a time?

Heck yes it would! So here we go:

Neat-o Enhanced Chunky Line Drawing Routine

First we need to be in the right mindframe. Let's say you're drawing a line where you move three pixels in x before it's time to take a step in y. Instead of plotting all three pixels it would of course be much more efficient to just stick a number like %00011100 in the drawing buffer. But somehow we need to keep track of a) how large the chunk needs to be, and b) where exactly the chunk is.

In the above example, we started at a particular x-value:

```
%00010000
```

and we want to keep adding ones to the right of the starting point; three, to be exact. Hmmm... we need to somehow rotate the starting bit in a way that leaves a trail of ones behind it. Maybe rotate and ORA with the original bit? But what happens when you take a step in Y?

No, we need something far sneakier. Let's say that instead of %00010000 we start with

```
x = %00011111
```

Now, with each step in the x direction, we do an arithmetic shift on x. So after one step we have

```
x = %00001111
```

and after two steps

```
x = %00000111
```

and at the third step of course

```
x = %00000011
```

Now it is time to take a step in Y. But now look: if we EOR x with its

original value xold = %00011111, we get

```
x EOR xold = %00011100
```

which is exactly the chunk we wanted. Moreover, x still remembers where it is, so we don't have to do anything special each time a step is taken in the y-direction.

So here is the algorithm for drawing a line in the x-direction:

```
initialize x, dx, etc.
xold = x
take a step in x: LSR X
have we hit the end of a column?  If so, then plot and check on y
is it time to take a step in y?
if not, take another step in x
if it is, then let a=x EOR xold
                plot a into the buffer
                let xold=x
keep on going until we're finished
```

This simple modification gives us a substantial speed increase -- on the old filled hires cube3d program, I measured a gain of one frame per second. Not earth-shattering, but not bad either! When faces are not filled, the difference is of course much more noticable.

There are a few things to be careful of. There was a bug in the old routine when the line was a single point. In that case dx=dy=0, and the program would draw a vertical line on the screen. There are probably some other things to be careful of, but since I wrote this part of the code three months ago I really don't remember any of them!

This takes care of horizontal line chunks -- what about vertical chunks? Well, because of the way points are plotted there is nothing we can do about them. But, as we shall soon see, if we use an EOR-buffer to fill faces we will be forced to take care of the vertical chunks!

General Polygon Routine

Now we can begin thinking about a general polygon routine. First we need a list of sets of points, where each set corresponds to a polygon. The first number in a set could be the number of (x,y,z) points in that set, and the points could then follow. So a triangle could be given by the data set:

```
3 -1 0 0 0 1 0 1 0 0
```

This would be a triangle with vertices at (-1,0,0), (0,1,0), and (1,0,0). We can mash a bunch of these sets together, but somehow we have to know when we've hit the end -- for this we can use a zero, since we don't want to plot polygons with zero points in them.

For that matter, how many points should there be in a polygon? There must be at least three, otherwise it makes no sense. Since we want our polygons to be closed, the computer should be smart enough to connect the last point to the first point -- in our triangle above, the computer would join (-1,0,0) to (0,1,0), (0,1,0) to (1,0,0), and (1,0,0) to (-1,0,0).

Now that we have a polygon, we want to rotate it. You will recall that we have calculated a rotation matrix M, which acts on points. So we need apply our rotation transform to each of the points in the polygon, i.e. multiply M times each point of the polygon. Furthermore, we need to project each of these points.

Uh-oh: matrix multiplication. In the past we have avoided this issue by putting the vertices of our cube at 1 or -1. So we need to use our multiplication routine from last time. But wait! As you recall, the last program used a specially modified multiplication table. To get a wider range of numbers to multiply we will need another set of multiplication tables -- no big whoop.

Now, if you review the multiplication routine from last time, it adds two numbers and subtracts two numbers. What kinds of numbers will we be dealing with? The matrix elements vary between -64..64. This then fixes our range of polygon coordinates from -64..64. Why? If the matrix element is 64, and we multiply it by 64, the multiplication routine will add 64 and 64 and get 128, which is right on the edge of our multiplication table.

Can we improve this rotation process in any way? In fact, we can cut down on the number of multiplications (i.e. do eight or even seven instead of nine multiplications). However, there is a fair amount of overhead involved in doing so, and our multiply routine is fast enough that the extra overhead and complexity really gain us very little in all but the most complicated of polygons. In other words, I didn't bother.

What about hidden faces? Again, from last time you may recall that a method was described which used the cross-product of the projected vectors. How do we implement this in the program? Well, if we take the first three points of the polygon, we have two vectors. Let's say these points are P1 P2 and P3. Then V1=P1-P2 and V2=P3-P2 are two vectors in the plane of the polygon which are connected at the point P2 (this analysis will of course only work if the polygon lies in some plane). Depending on how we take the cross product, the sign will be positive or negative, and this will tell us if the polygon is visible.

Depending on how we take the cross product? Absolutely. v1 x v2 = -v2 x v1. What it really boils down to is how you define the points in your polygon. Specifically, what order they are in. Points that are specified in a clockwise manner will give a face pointing in the opposite direction of a polygon with the same points specified in a counter-clockwise order. In my program, the polygons must be entered in counter-clockwise order (with you facing the polygon) for hidden faces to work the way you want them to ;-).

One other neat thing to have is the ability to zoom in and out. We know from the very first article that zooming corresponds to multiplying the projected points by a number, so that's what we'll do. The multiplication routine returns A=A*Y/64, so a zoom factor of 64 would be like multiplying the point by one. All the program does is multiply the projected points by a number zoom, unless zoom=64, in which case the program skips the zoom multiply. Be warned! No checks of any sort are made in the program, so you can zoom at your own risk!

The important things to remember are: when entering polygons, make sure the numbers range from -64 to 64, and that you enter points in counterclockwise. Our triangle example above really should have been entered as, say,

```
3 -64 0 0 64 0 0 0 64 0
```

Filled Faces -- Using an EOR buffer

Well we still have one thing left, which was alluded to in the

previous article: using EOR to make a filled face. Some possible difficulties were raised, but when you plot a single polygon at a time, the problem becomes vastly simplified.

First I should perhaps remind you what exclusive-or is: either A or B, but not both. So 1 EOR 0 = 1, as does 0 EOR 1, but 0 EOR 0 = 0 and 1 EOR 1 = 0. As a simple introduction to using this for filling faces, consider the following piece of the drawing buffer:

```
00001011 M1
00000000 M2
00000001 M3
00001010 M4
```

Lets say we move down memory, EORing as we go. Let M2 = M1 EOR M2. Then let M3 = M2 EOR M3. Then let M4 = M3 EOR M4. Our little piece of memory is now:

```
00001011 M1
00001011 M2
00001010 M3
00000000 M4
```

What just happened? Imagine that the original memory was a series of pieces of line segments. We have just filled in the area between the two line segments, like magic!

If you still aren't getting it, draw a large section of memory, and then draw an object in it, like a triangle, or a trapazoid, and EOR the memory by hand, starting from the top and moving downwards.

EOR flips bits. If you start with a zero, it stays zero until it hits a one. It will then stay one until it hits another one. So you can see that if you have an object bounded by ones, EORing successive memory locations will automagically fill the object.

Right? Well, we have to be careful. One major problem is a vertical line:

```
1                       1
1       goes to         0
1                       1
1                       0
```

Not only is the resultant line dashed, but if there are an odd number of points in the line segment, the last one will happily move downwards in memory, and give you a much longer vertical line than you expected! Since any line with slope greater than one is made up of a series of line segments, this is a major consideration.

Another problem arises with single points: a one just sitting all by itself will also generate a nice streak down your drawing area.

If you think about it, what we ideally want to have is an object that at any given value of x there are exactly two points, one defining the top of the object, and the other defining the bottom. This gives us the insight to solve the above two problems.

First let's think about vertical lines. In principle we could plot the first and last endpoints of each vertical line chunk, but that is exactly what we don't want! Remember that these are closed polygons, which means that there are _two_ lines we need to think about. If I plot just a single point in each vertical line segment, there must be another point somehwere, either above or below it, from another line segment, which will close the point to EOR-filling. Remember, we want exactly two points at each value of x: one will come from the line, and the other will come from the other line which must lie above or below the current one.

Furthermore, with any convex polygon there are exactly two lines which come together at each vertex of the polygon. This means that there are only certain cases which we need to worry about. For instance, two lines might join in any of the following ways:

```
\                   /    \    /
 \                 /      \  /
  \_____    _____/         \/  etc.
```

If you draw out the different cases involving vertical lines, you can see that you have to be careful about plotting the lines. One tricky one is where two vertical lines with different slopes overlap at the point of intersection.

So after staring at these pictures for a while, you can find a consistent method which solves these difficulties. As long as you follow the following rules, the problems all disappear; the line routine needs to be modified slightly:

1) When plotting a vertical line (i.e. big steps in Y direction), don't plot the endpoints (i.e. x1,y1 and x2,y2).

2) When plotting a vertical line, consistently plot either the first part of each chunk or the last part of each chunk (excluding the endpoints of course). In other words, only plot a point when you take a step in x, and then plot one and only one point.

Now I deduced these by staring at pictures for a few hours and trying different things like top/bottom of chunk, left/right, first/last, etc. You can see that in some cases this ensures that only one point appears on a given line segment. But to me the only way to convince yourself that this really does work is to draw a bunch of pictures, and try it out! You have cases where two vertical lines intersect, and where a vertical line intersects a horizontal line.

But there is still one thing which we have forgotten -- the case of a single point. This can happen in, for instance, a pointy triangle, pointing in the x-direction. How do we fix this? By simply avoiding the point: in the line drawing routine, use EOR to plot the points instead of ORA. Since vertical lines skip the endpoints, vertical-horizontal intersections are OK. Horizontal-horizontal intersections will force the point of intersection to be zero.

Uh-oh, what about intersections like -----*------. Quite frankly I just thought of it, and I think my program will fail on intersections like these. Drat. Well, that just gives us something for next time!

One other thing needs to be mentioned: for EOR-filling to be useful you need to draw the polygon in a special buffer, and then EOR this buffer into the main display buffer. If you try to EOR the display buffer directly you are going to have a whole heap of trouble, such as the concerns raised last time.

Finally, this gives a simple way of filling with patterns instead of boring monocolor. Instead of EOR (EORBUF),Y : ORA (DRAWBUF),Y you can use EOR (EORBUF),Y : AND PATTERN,Y : ORA (DRAWBUF),Y (as long as you preserve the original EOR (EORBUF),Y).

Well I am extremely tired and I hope Craig hasn't sent out C=Hacking without me! I hope you have fun playing with the program, and I would be very interested in seeing any neat geometric shapes you might design!

Program notes:

- Hidden faces defaults to "on". If you enter a shape and a black screen comes up, hit 'h' to turn off hidden faces (you probably entered the polygon clockwise).
- There is no pattern filling -- just simple EOR with a twist: the EOR buffer is EOR'd into the drawing buffer.
- You might start hosing memory if you zoom too large.

SLJ 6/15/95

Addendum

Stephen Judd sjudd@nwu.edu

Last time we put a circle into the 2D graphics toolbox. Chris McBride has pointed something out to me about the algorithm, which makes it complete. As you may recall, the algorithm gave a very squarish circle for small radii. Chris told me that setting the initial counter value to R/2, instead of R, gave a perfect circle. What is going on? If you recall the algorithm, we are computing a fractional quantity, and when that quantity becomes larger than one, we decrease X. Wouldn't it be a whole lot smarter to round off that fraction instead of truncate it? Of course it would, and that is what starting the counter at R/2 does.

So, to update the previous algorithm, A should be initialized to R/2 instead of R, which means that we change

```
        LDA R
```

to

```
        LDA R
        LSR
```

for a perfect circle every time.

```
begin 666 cube3d3.2.s
M ' J*BHJ*BHJ*BHJ*BHJ*BHJ*BHJ*BHJ*BHJ*BHJ*BHJ*@TJH*"@H*"@H*"@
MH*"@H*"@H*"@H*"@H*"@H*"@H*"@*@TJH'-415!(14Z@:E5$1*"@H*"@H*"@
MH*"@H*"@H*"@*@TJH&=%3U)'1:!T05E,3U*@H*"@H*"@H*"@H*"@H*"@*@TJ
MH'-405)4140ZH#<O,3$O.32@H*"@H*"@H*"@H*"@*@TJH&9)3DE32$5$.J W
```

```
M+S$Y+SDTH*"@H*"@H*"@H*"@*@TJH%8R+C"@8T]-4$Q%5$5$.J Q,B\Q-R\Y
M-*"@H*"@*@TJH%8S+C"@8T]-4$Q%5$5$.J S+S(P+SDUH*"@H*"@*@TJH%8S
M+C&@8T]-4$Q%5$5$.J V+S$T+SDUH*"@H*"@*@TJH%8S+C*@8T]-4$Q%5$5$
M.J V+S$U+SDUH*"@H*"@*@TJH*"@H*"@H*"@H*"@H*"@H*"@H*"@H*"@H*"@
MH*"@*@TJH'=%3$PLH$E&H$%,3*!'3T53H%=%3$R@5$A)4Z"@*@TJH%!23T=2
M04V@5TE,3*!23U1!5$6@0:!#54)%+J"@*@TJH*"@H*"@H*"@H*"@H*"@H*"@
MH*"@H*"@H*"@H*"@*@TJH%8R+C"@*Z!N15>@04Y$H&E-4%)/5D5$(:"@H*"@
M*@TJH&Y/5Z!7251(H$9!4U1%4J!23U5424Y%4RR@H*"@*@TJH$A)1$1%3J!3
M55)&04-%4RR@1DE,3$5$H*"@H*"@*@TJH$9!0T53+*!!3D2@15A44D&@5$]0
MH%-%0U)%5*"@*@TJH%1%6%2@34534T%'15,AH*"@H*"@H*"@H*"@H*"@*@TJ
MH*"@H*"@H*"@H*"@H*"@H*"@H*"@H*"@H*"@*@TJH%8S+C"@*Z!F05-4
MH$-(54Y+6:!,24Y%H*"@H*"@*@TJH%)/551)3D4NH*"@H*"@H*"@H*"@H*"@
MH*"@H*"@*@TJH*"@H*"@H*"@H*"@H*"@H*"@H*"@H*"@*@TJH%8S
M+C&@*Z!G14Y%4D%,H%!/3%E'3TZ@4$Q/5*"@*@TJH%=)5$B@2$E$1$5.H$9!
M0T53H"AX+5!23T150U0I*@TJH$%.1*!:3T]-H$9%05154D4NH*"@H*"@H*"@
MH*"@*@TJH*"@H*"@H*"@H*"@H*"@H*"@H*"@H*"@H*"@H*"@*@TJH%8S+C*@
M*Z!E;W(M0E5&1D52H$9)3$Q)3D>@H*"@*@TJH*"@H*"@H*"@H*"@H*"@H*"@
MH*"@H*"@H*"@H*"@*@TJH'1(25.@4%)/1U)!3:!)4Z!)3E1%3D1%1*!43Z"@
M*@TJH$%#0T]-4$%.6:!42$6@05)424-,1:!)3J"@H*"@*@TJH&,]:$%#2TE.
M1RR@:E5.+J Y-:!)4U-512Z@H*"@*@TJH&9/4J!$151!24Q3H$].H%1(25.@
M4%)/1U)!32R@*@TJH%)%042@5$A%H$%25$E#3$4AH*"@H*"@H*"@H*"@*@TJ
MH*"@H*"@H*"@H*"@H*"@H*"@H*"@H*"@H*"@H*"@*@TJH'=2251%H%1/H%53
M(:"@H*"@H*"@H*"@H*"@H*"@*@TJH*"@H*"@H*"@H*"@H*"@H*"@H*"@H*"@
MH*"@H*"@*@TJH&U94T5,1J!72$5.H%E/54Y'H$1)1*"@H*"@H*"@*@TJH$5!
M1T523%F@1E)%455%3E2@H*"@H*"@H*"@H*"@*@TJH&1/0U1/4J!!3D2@<T%)
M3E0LH$%.1*!(14%21*"@*@TJH$=214%4H&%21U5-14Y4H*"@H*"@H*"@H*"@
MH*"@*@TJH*!A0D]55*!)5*!!3D2@04)/550ZH$)55*"@H*"@*@TJH*!%5D52
M34]21:"@H*"@H*"@H*"@H*"@H*"@H*"@*@TJH&-!346@3U54H$)9H%1(1:!3
M04U%H&1/3U*@H*"@*@TJH$%3H$E.H&F@5T5.5"Z@H*"@H*"@H*"@H*"@H*"@
M*@TJH*"@H"V@<E5"04E9052@H*"@H*"@H*"@H*"@H*"@*@TJH*"@H*"@H*"@
MH*"@H*"@H*"@H*"@H*"@*@TJH'1(3U5'2*!IH%-014%+H%=)5$B@
M5$A%H*"@H*"@*@TJH%1/3D=515.@3T:@345.H$%.1*!/1J!!3D=,15.@*@TJ
MH$%.1*!(059%H$Y/5*!,3U9%+*!IH$%-H*"@H*"@*@TJH$)%0T]-1:!!4Z!3
M3U5.1$E.1Z!"4D%34RR@3U*@*@TJH$&@5$E.2TQ)3D>@0UE-0D%,+J"@H*"@
MH*"@H*"@*@TJH*"@H"V@,:!C3U))3E1(24%.4Z Q,Z"@H*"@H*"@*@TJH*"@
MH*"@H*"@H*"@H*"@H*"@H*"@H*"@H*"@*@TJH' N<RZ@=$A)4Z!705.@
M5U))5%1%3J!54TE.1Z"@*@TJH*"@H*"@;4523$E.H#$R."Z@H*"@H*"@H*"@
MH*"@*@TJ*BHJ*BHJ*BHJ*BHJ*BHJ*BHJ*BHJ*BHJ*BHJ*@T-(&]R9R D
M.# P, T-*J!C3TY35$%.5%,-#6)U9F8Q(&5Q=2 D,S P," [9DE24U2@0TA!
M4D%#5$52H%-%5 UB=69F,B!E<74@)#,X,# @.W-%0T].1*!#2$%204-415*@
M4T54#65O<F)U9B!E<74@)#0P,# @.V5O<BU"549&15(-8G5F9F5R(&5Q=2 D
M83,@.W!215-534%"3%F@5$A%H%1!4$6@5T].)U2@0D6@4E5.3DE.1PUX,2!E
M<74@)&9B(#MP3TE.5%.@1D]2H$1205=)3D>@0:!,24Y%#7DQ(&5Q=2 D9F,@
M.W1(15-%H%I%4D^@4$%'1:!!1$1215-315,->#(@97%U("1F9" [1$].)U2@
M0T].1DQ)0U2@5TE42*!B87-I8PUY,B!E<74@)&9E#6]L9'@@97%U("1F9 UC
M:'5N:R!E<74@)&9E#61X(&5Q=2 D-C<@.W1(25.@25.@4TA!4D5$H%=)5$B@
M=#&@0D5,3U<-9'D@97%U("0V. UT96UP,2!E<74@)&9B(#MO1J!#3U524T4L
MH$-/54Q$H$-/3D9,24-4H%=)5$B@6#$-=&5M<#(@97%U("1F8R [=$5-4$]2
M05)9H%9!4DE!0DQ%4PUZ=&5M<"!E<74@)# R(#MU4T5$H$9/4J!"549&15*@
M4U=!4"Z@H&1/3B=4H%1/54-(+@UZ,2!E<74@)#(R(#MU4T5$H$)9H$U!5$B@
M4D]55$E.10UZ,B!E<74@)#(T(#MD3TXG5*!43U5#2*!42$531:!%251(15(A
M#7HS(&5Q=2 D,C8->C0@97%U("0R. UK(&5Q=2 D8C8@.V-/3E-404Y4H%53
M142@1D]2H$A)1$1%3@T@(" [4U521D%#1:!$151%0U1)3TZ@+:!$3TXG5*!4
M3U5#2 UH:61E(&5Q=2 D8C4@.V%21:!355)&04-%4Z!(241$14X_#69I;&P@
M97%U("0U," [85)%H%=%H%5324Y'H&5O<BU&24Q,/PUA;F=M87@@97%U(#$R
M," [=$A%4D6@05)%H#(J4$DO04Y'34%8H$%.1TQ%4PT-*J!V:6,-#79M8W-B
M(&5Q=2 D9# Q. UB:V=N9"!E<74@)&0P,C -8F]R9&5R(&5Q=2 D9# R,0US
M<W1A<G0@97%U(#$S-#0@.U)/5Z YH$E.H%-#4D5%3J!-14U/4EF@052@,3 R
M- T-#2J@:T523D%,#0UC:')O=70@97%U("1F9F0R#6=E=&EN(&5Q=2 D9F9E
M- T-*J!S3TU%H%9!4DE!0DQ%4PT-9VQO8GAM:6X@/2 D,V8@.W1(15-%H$%2
M1:!54T5$H$E.H$-,14%224Y'H%1(10UG;&]B>&UA>" ]("0T," [1%)!5TE.
M1Z H1TQ/0D%,*:!"549&15(-9VQO8GEM:6X@/2 D-#$-9VQO8GEM87@@/2 D
M-#(-;&]C>&UI;B ]("0U-R [=$A%4T6@05)%H%53142@24Z@0TQ%05))3D>@
M5$A%#6QO8WAM87@@/2 D-3@@.V5O<J H3$]#04PIH$)51D9%4@UL;V-Y;6EN
M(#T@)#4Y#6QO8WEM87@@/2 D-C -<#%X(#T@)#DR(#MT2$531:!!4D6@5$5-
M4$]205)9H%-43U)!1T4-<#%Y(#T@)#DS(#MU4T5$H$E.H%!,3U1424Y'H%1(
M1:!04D]*14-424].#7 Q>B ]("0Y- UP,G@@/2 D.34@.W1(15F@05)%H$A%
M4D6@4T^@5$A!5*!710UP,GD@/2 D.38@.T1/3B=4H$A!5D6@5$^@4D5#04Q#
M54Q!5$6@5$A%32X-<#)Z(#T@)&%E#7 S>" ]("1A9B [=$A%6:!-04M%H$Q)
M1D6@14%362X-<#-Y(#T@)&(P#7 S>B ]("1B,2 [=TA9H$%21:!93U6@3$]/
M2TE.1Z!!5*!-1:!,24M%H%1(050_#7 Q=" ]("1B,B [9$].)U2@64]5H%12
M55-4H$U%/PUP,G0@/2 D8C,-<#-T(#T@)&(T(#MH059)3D>@04Y/5$A%4J!#
M2$E,1*!705-.)U2@35F@241%02X-:6YD97@@/2 D-3$-8V]U;G1P=',@/2 D
M-3(->F]O;2 ]("0W,2 [>D]/3:!&04-43U(-9'-X(#T@)#8Q(#MD<WB@25.@
M5$A%H$E.0U)%345.5*!&3U(-(" @.U)/5$%424Y'H$%23U5.1*!8#61S>2 ]
M("0V,B [<TE-24Q!4J!&3U*@9'-Y+*!D<WH-9'-Z(#T@)#8S#7-X(#T@)#8T
M(#MT2$531:!!4D6@5$A%H$%#5%5!3*!!3D=,15.@24Z@6*!9H$%.1*!:#7-Y
M(#T@)#8U#7-Z(#T@)#8V#70Q(#T@)#8W(#MT2$531:!!4D6@55-%1*!)3J!4
M2$6@4D]4051)3TX-=#(@/2 D-C@-=#,@/2 D-CD@.W-%1:!42$6@05)424-,
M1:!&3U*@34]21:!$151!24Q3#70T(#T@)#9A#70U(#T@)#9B#70V(#T@)#9C
M#70W(#T@)#9D#70X(#T@)#9E#70Y(#T@)#9F#70Q," ]("0W, UA,3$@/2 D
```

```
M834@.W1(15-%H$%21:!42$6@14Q%345.5%.@3T:@5$A%H%)/5$%424].H$U!
M5%))6 UB,3(@/2 D838@.WAY>@UC,3,@/2 D83<-9#(Q(#T@)&$X(#MT2$6@
M3E5-0D52H$1%3D]415.@*%)/5RQ#3TQ534XI#64R,B ]("1A.0UF,C,@/2 D
M86$-9S,Q(#T@)&%B#6@S,B ]("1A8PUI,S,@/2 D860-#0TJ*BJ@;4%#4D]3
M#0UM;W9E(&UA8PT@;&1A(%TQ#2!S=&$@73(-(#P\/ T-9V5T:V5Y(&UA8R @
M.W=!252@1D]2H$&@2T594%)%4U,-=V%I="!J<W(@9V5T:6X-(&-M<" C,# -
M(&)E<2!W86ET#2 \/#P-#61E8G5G(&UA8R @.W!224Y4H$&@0TA!4D%#5$52
M#2!D;Z PH* [9$].)U2@05-314U"3$4-#2!L9&&@(UTQ#2!J<W*@8VAR;W5T
M#2!C;&D-(#X^/B!G971K97D@.V%.1*!704E4H%1/H$-/3E1)3E5%#2!C;7 @
M(R=3)R [;5F@4T5#4D5#5*!35TE40TB@2T59#2!B;F4@;#$-(&IS<B!C;&5A
M;G5P#2!J;7 @9&]N90UL,2!C;7 @(R=8)R [;5F@4T5#4D54H$%"3U)4H$M%
M60T@8FYE(&1O;F4-(&IM<"!C;&5A;G5P#2!F:6X-9&]N92 \/#P-#61E8G5G
M82!M86,-(&1OH# -(&QD82!=,0T@<W1A(#$P,C0-(&9I;@UD;VYE82 \/#P-
M#2HM+2TM+2TM+2TM+2TM+2TM+2TM+2TM+2TM#0T@;&1A(",D,# -
M('-T82!B:V=N9 T@<W1A(&)O<F1E<@T@;&1A('9M8W-B#2!A;F0@(R4P,# P
M,3$Q,2 [<T-2145.H$U%34]26:!43Z Q,#(T#2!O<F$@(R4P,# Q,# P, T@
M<W1A('9M8W-B#0T@;&1Y(",P, T@;&1A(",\='1E>'0-('-T82!T96UP,0T@
M;&1A(",^='1E>'0-('-T82!T96UP,@T@:FUP('1I=&QE#71T97AT(&AE>" Y
M,S U,3$Q,3$Q(#M#3$5!4J!30U)%14XLH%=(251%+*!#4E-2H$1.#2!T>'0@
M)Z"@H*"@H*"@H*"@H*!#54)%,T2@5C,N,B<L,$0L,$0-('1X=" GH*"@H*"@
MH*"@H*"@H*"@H*"@0EDG+#!$#2!H97@@.68@.T-904X-('1X=" GH*"@H%-4
M15!(14Z@2E5$1"<-(&AE>" Y.0T@='AT(">@H*"@1T5/4D=%H%1!64Q/4B<L
M,$0L,$0-(&AE>" Y8@T@='AT(">@H$-(14-+H$]55*!42$6@2D%.+J Y-:!)
M4U-51:!/1B<L,$0-(&AE>" Y-@T@='AT(">@H$,]2$%#2TE.1R<-(&AE>" Y
M8@T@='AT(">@1D]2H$U/4D6@1$5404E,4R$G+#!$#2!H97@@,&0Q9#%D.64Q
M,@T@='AT("=&,2]&,B<L.3(-('1X=" GH"V@24Y#+T1%0Z!8+5)/5$%424].
M)RPP1 T@:&5X(#%D,60Q,@T@='AT("=&,R]&-"<L.3(-('1X=" GH"V@24Y#
M+T1%0Z!9+5)/5$%424].)RPP1 T@:&5X(#%D,60Q,@T@='AT("=&-2]&-B<L
M.3(-('1X=" GH"V@24Y#+T1%0Z!:+5)/5$%424].)RPP1 T@:&5X(#%D,60Q
M,@T@='AT(">@1C>@H"<L.3(-('1X=" GH"V@4D53150G+#!$#2!H97@@,60Q
M9#$R#2!T>'0@)Z K+RV@)RPY,@T@='AT(">@+:!:3T]-H$E.+T]55"<L,$0-
M(&AE>" Q9#%D,3(-('1X=" GH*!(H* G+#DR#2!T>'0@)Z MH%1/1T=,1:!(
M241$14Z@4U521D%#15,G+#!$#2!H97@@,60Q9#$R#2!T>'0@)U-004-%)RPY
M,@T@='AT(">@+:!43T='3$6@4U521D%#1:!&24Q,24Y')RPP1"PP1 T@='AT
M(">@H%!215-3H%&@5$^@455)5"<L,$0-(&AE>" P9# U#2!T>'0@)Z"@H*"@
MH%!215-3H$%.6:!+15F@5$^@0D5'24XG+#!$#2!H97@@,# -=&ET;&4@;&1A
M("AT96UP,2DL>0T@8F5Q(#IC;VYT#2!J<W(@8VAR;W5T#2!I;GD-(&)N92!T
M:71L90T@:6YC('1E;7 R#2!J;7 @=&ET;&4-.F-O;G0@/CX^(&=E=&ME>0T-
M*BHJ*J!S152@55"@5$%"3$53*#\I#0TJH'1!0DQ%4Z!!4D6@0U524D5.5$Q9
MH%-%5*!54*!)3J!B87-I8PTJH$%.1*!"6:!42$6@05-314U"3$52+@T-=&%B
M;&5S(&QD82 C/G1M871H,0T@<W1A('HQ*S$-('-T82!Z,BLQ#2!L9&$@(SYT
M;6%T:#(-('-T82!Z,RLQ#2!S=&$@>C0K,0T-*BHJ*J!C3$5!4J!30U)%14Z@
M04Y$H%-%5*!54* B0DE434%0(@US971U<"!L9&$@(R0P,2 [=TA)5$4-('-T
M82 D9# R,2 [=$A)4Z!)4Z!$3TY%H%-/H%1(052@3TQ$15(-(&QD82 C,30W
M(#M-04-(24Y%4Z!724Q,H%-%5*!54 T@:G-R(&-H<F]U= T@;&1A(",D,# @
M.T-/4E)%0U1,60T@<W1A("1D,#(Q#2!L9&$@(SQS<W1A<G0-(&%D8R C,3(@
M.W1(1:!'3T%,H$E3H%1/H$-%3E1%4J!42$6@1U)!4$A)0U,-('-T82!T96UP
M,2 [8T],54U.H#$R#2!L9&$@(SYS<W1A<G0@.W)/5Z Y#2!S=&$@=&5M<#$K
M,2 [<W-T87)TH%!/24Y44Z!43Z!23U>@.0T@;&1A(",P, T@;&1Y(",P, T@
M;&1X(",P," [6*!724Q,H$-/54Y4H#$VH%)/5U.@1D]2H%53#2!C;&,-#3IL
M;V]P('-T82 H=&5M<#$I+'D-(&EN>0T@861C(",Q-@T@8F-C(#IL;V]P#2!C
M;&,-(&QD82!T96UP,0T@861C(",T," [;D5%1*!43Z!!1$2@-#"@5$^@5$A%
MH$)!4T6@4$])3E1%4@T@<W1A('1E;7 Q(#MT3Z!*54U0H%1/H%1(1:!.15A4
MH%)/5PT@;&1A('1E;7 Q*S$-(&%D8R C,# @.W1!2T6@0T%21:!/1J!#05)2
M2453#2!S=&$@=&5M<#$K,0T@;&1Y(",P, T@:6YX#2!T>&$@(#MXH$E3H$%,
M4T^@04Z@24Y$15B@24Y43Z!42$6@0TA!4D%#5$52H$Y534)%4@T@8W!X(",Q
M-@T@8FYE(#IL;V]P(#MN145$H%1/H$1/H$E4H#$VH%1)3453#0TJ*BHJH&-,
M14%2H$)51D9%4E,-#2!L9&$@(SQB=69F,0T@<W1A(&)U9F9E<@T@;&1A(",^
M8G5F9C$-('-T82!B=69F97(K,0T@;&1Y(",D,# -(&QD>" C,C0@.V%34U5-
M24Y'H$%,3*!42%)%1:!"549&15)3H$%210T@;&1A(",D,# @.T)!0TLM5$\M
M0D%#2PTZ8FQO;W @<W1A("AB=69F97(I+'D-(&EN>0T@8FYE(#IB;&]O< T@
M:6YC(&)U9F9E<BLQ#2!D97@-(&)N92 Z8FQO;W -#2HJ*BJ@<T54H%50H$)5
M1D9%4E,-#2!L9&$@(SQB=69F,0T@<W1A(&)U9F9E<@T@;&1A(",^8G5F9C$-
M('-T82!B=69F97(K,0T@<W1A('IT96UP(#M:5$5-4*!724Q,H$U!2T6@3$E&
M1:!324U03$6@1D]2H%53#2!L9&$@=FUC<V(-(&%N9" C)3$Q,3$P,# Q(#MS
M5$%25*!(15)%H%-/H%1(052@4U=!4*!"549&15)3H%=)3$R@5T]22Z!224=(
M5 T@;W)A(",E,# P,#$Q,3 -('-T82!V;6-S8@T-*BHJ*J!S152@55"@24Y)
M5$E!3*!604Q515,-#6EN:70@;&1A(",P, T@<W1A(&QO8WAM:6X-('-T82!L
M;V-X;6%X#2!S=&$@;&]C>6UI;@T@<W1A(&QO8WEM87@-('-T82!G;&]B>&UI
M;@T@<W1A(&=L;V)Y;6EN#2!S=&$@9VQO8GAM87@-('-T82!G;&]B>6UA> T@
M<W1A(&1S> T@<W1A(&1S>0T@<W1A(&1S>@T@<W1A('-X#2!S=&$@<WD-('-T
M82!S>@T@<W1A(&9I;&P-(&QD82 C,#$-('-T82!H:61E#2!L9&$@(S8T#2!S
M=&$@>F]O;0T-*BTM+2TM+2TM+2TM+2TM+2TM+2TM+2TM+2TM+2T-*J!M
M04E.H$Q/3U -#2HJ*BJ@9T54H$M%65!215-3#0UM86EN#2!C;&D-:W!R97-S
M(&IS<B!G971I;@T@8VUP(",Q,S,@.V8Q/PT@8FYE(#IF,@T@;&1A(&1S> T@
M8VUP("-A;F=M87@O,B [;D^@34]21:!42$%.H%!)#2!B97$@.F-O;G0Q#2!I
M;F,@9'-X(#M/5$A%4E=)4T6@24Y#4D5!4T6@6"U23U1!5$E/3@T@:FUP(#IC
M;VYT#3IF,B!C;7 @(S$S-R [9C(_#2!B;F4@.F8S#2!L9&$@9'-X#2!B97$@
M.F-O;G0Q#2!D96,@9'-X#2!J;7 @.F-O;G0-.F8S(&-M<" C,3,T#2!B;F4@
M.F8T#2!L9&$@9'-Y#2!C;7 @(V%N9VUA>"\R#2!B97$@.F-O;G0Q#2!I;F,@
M9'-Y(#MI3D-214%31:!9+5)/5$%424].#2!J;7 @.F-O;G0-.F8T(&-M<" C
```

```
M,3,X#2!B;F4@.F8U#2!L9&$@9'-Y#2!B97$@.F-O;G0Q#2!D96,@9'-Y#2!J
M;7 @.F-O;G0-.F8U(&-M<" C,3,U#2!B;F4@.F8V#2!L9&$@9'-Z#2!C;7 @
M(V%N9VUA>"\R#2!B97$@.F-O;G0Q#2!I;F,@9'-Z(#M:+5)/5$%424].#2!J
M;7 @.F-O;G0-.F8V(&-M<" C,3,Y#2!B;F4@.F8W#2!L9&$@9'-Z#2!B97$@
M.F-O;G0Q#2!D96,@9'-Z#2!J;7 @.F-O;G0-.F8W(&-M<" C,3,V#2!B;F4@
M.G!L=7,-(&IM<"!I;FET#3IC;VYT,2!J;7 @.F-O;G0-.G!L=7,@8VUP(",G
M*R<-(&)N92 Z;6EN=7,-(&EN8R!Z;V]M(#MB04@LH%=(3Z!.145$4Z!%4E)/
M4J!#2$5#2TE.1S\-(&EN8R!Z;V]M#2!J;7 @.F-O;G0-.FUI;G5S(&-M<" C
M)RTG#2!B;F4@.F@-(&1E8R!Z;V]M#2!D96,@>F]O;0T@8G!L(#IC;VYT#2!I
M;F,@>F]O;0T@:6YC('IO;VT-(&IM<" Z8V]N= TZ:"!C;7 @(R=()PT@8FYE
M(#IS<&%C90T@;&1A(&AI9&4-(&5O<B C)# Q#2!S=&$@:&ED90T@:FUP(#IC
M;VYT#3IS<&%C92!C;7 @(R>@)PT@8FYE(#IQ#2!L9&$@9FEL; T@96]R(",D
M,#$-('-T82!F:6QL#2!J;7 @.F-O;G0-.G$@8VUP(",G42<@.U&@455)5%,-
M(&)N92 Z8V]N= T@:FUP(&-L96%N=7 -#3IC;VYT('-E:2 @.W-0145$H%1(
M24Y'4Z!54*!!H$))5 T-*BHJ*J!U4$1!5$6@04Y'3$53#0UU<&1A=&4@8VQC
M#2!L9&$@<W@-(&%D8R!D<W@-(&-M<" C86YG;6%X(#MA4D6@5T6@/CV@34%8
M24U53:!!3D=,13\-(&)C8R Z8V]N=#$-('-B8R C86YG;6%X(#II1B!33RP@
M4D53150-.F-O;G0Q('-T82!S> T@8VQC#2!L9&$@<WD-(&%D8R!D<WD-(&-M
M<" C86YG;6%X#2!B8V,@.F-O;G0R#2!S8F,@(V%N9VUA>" [<T%-1:!$14%,
M#3IC;VYT,B!S=&$@<WD-(&-L8PT@;&1A('-Z#2!A9&,@9'-Z#2!C;7 @(V%N
M9VUA> T@8F-C(#IC;VYT,PT@<V)C("-A;F=M87@-.F-O;G0S('-T82!S>@T-
M*BHJ*J!R3U1!5$6@0T]/4D1)3D%415,-#7)O=&%T90T-*BHJH&9)4E-4+*!#
M04Q#54Q!5$6@5#$L5#(L+BXN+%0Q, T-*BJ@=%=/H$U!0U)/4Z!43Z!324U0
M3$E&6:!/55*@3$E&10UA9&1A(&UA8R @.V%$1*!45T^@04Y'3$53H%1/1T54
M2$52#2!C;&,-(&QD82!=,0T@861C(%TR#2!C;7 @(V%N9VUA>" [:5.@5$A%
MH%-53: ^H#(J4$D_#2!B8V,@9&]N90T@<V)C("-A;F=M87@@.VE&H%-/+*!3
M54)44D%#5* R*E!)#61O;F4@/#P\#0US=6)A(&UA8R @.W-50E1204-4H%17
M3Z!!3D=,15,-('-E8PT@;&1A(%TQ#2!S8F,@73(-(&)C<R!D;VYE#2!A9&,@
M(V%N9VUA>" [;T]04RR@5T6@3D5%1*!43Z!!1$2@,BI020UD;VYE(#P\/ T-
M*BJ@;D]7H$-!3$-53$%41:!4,2Q4,BQ%5$,N#0T@/CX^('-U8F$L<WD[<WH-
M('-T82!T,2 [5#$]4UDM4UH-(#X^/B!A9&1A+'-Y.W-Z#2!S=&$@=#(@.U0R
M/5-9*U-:#2 ^/CX@861D82QS>#MS>@T@<W1A('0S(#M4,SU36"M36@T@/CX^
M('-U8F$L<W@[<WH-('-T82!T-" [5#0]4U@M4UH-(#X^/B!A9&1A+'-X.W0R
M#2!S=&$@=#4@.U0U/5-8*U0R#2 ^/CX@<W5B82QS>#MT,0T@<W1A('0V(#M4
M-CU36"U4,0T@/CX^(&%D9&$L<W@[=#$-('-T82!T-R [5#<]4U@K5#$-(#X^
M/B!S=6)A+'0R.W-X#2!S=&$@=#@@.U0X/50R+5-8#2 ^/CX@<W5B82QS>3MS
M> T@<W1A('0Y(#M4.3U362U36 T@/CX^(&%D9&$L<W@[<WD-('-T82!T,3 @
M.U0Q,#U36"M360T-*J!E5*!63TE,02$-#2HJ*J!N15A4+*!#04Q#54Q!5$6@
M82QB+&,L+BXN+&D-#2HJH&%.3U1(15*@55-%1E5,H$Q)5%1,1:!-04-23PUD
M:78R(&UA8R @.V1)5DE$1:!!H%-)1TY%1*!.54U"15*@0EF@,@T[:52@25.@
M05-354U%1*!42$%4H%1(1:!.54U"15(-(&)P;"!P;W,@.TE3H$E.H%1(1:!!
M0T-5355,051/4@T@8VQC#2!E;W(@(R1F9B [=T6@3D5%1*!43Z!53BU.14=!
M5$E61:!42$6@3E5-0D52#2!A9&,@(S Q(#M"6:!404M)3D>@250G4Z!#3TU0
M3$5-14Y4#2!L<W(@(#M$259)1$6@0EF@5%=/#2!C;&,-(&5O<B C)&9F#2!A
M9&,@(S Q(#MM04M%H$E4H$Y%1T%4259%H$%'04E.#2!J;7 @9&]N961I=@UP
M;W,@;'-R(" [;E5-0D52H$E3H%!/4TE4259%#61O;F5D:78@/#P\#0UM=6PR
M(&UA8R @.VU53%1)4$Q9H$&@4TE'3D5$H$Y534)%4J!"6: R#2!B<&P@<&]S
M;0T@8VQC#2!E;W(@(R1F9@T@861C(",D,#$-(&%S; T@8VQC#2!E;W(@(R1F
M9@T@861C(",D,#$-(&IM<"!D;VYE;75L#7!O<VT@87-L#61O;F5M=6P@/#P\
M#0TJ*J!N3U1%H%1(052@5T6@05)%H$-54E)%3E1,6:!-04M)3D>@0:!-24Y/
M4J!,14%0#2HJH$]&H$9!251(H%1(052@3D^@3U9%4D9,3U=3H%=)3$R@3T-#
M55(N#0TZ8V%L8V$@8VQC#2!L9'@@=#$-(&QD82!C;W,L> T@;&1X('0R#2!A
M9&,@8V]S+'@-('-T82!A,3$@.V$]*$-/4RA4,2DK0T]3*%0R*2DO,@TZ8V%L
M8V(@;&1X('0Q#2!L9&$@<VEN+'@-('-E8PT@;&1X('0R#2!S8F,@<VEN+'@-
M('-T82!B,3(@.V(]*%-)3BA4,2DM4TE.*%0R*2DO,@TZ8V%L8V,@;&1X('-Y
M#2!L9&$@<VEN+'@-(#X^/B!M=6PR#2!S=&$@8S$S(#MC/5-)3BA362D-.F-A
M;&-D('-E8PT@;&1X('0X#2!L9&$@8V]S+'@-(&QD>"!T-PT@<V)C(&-O<RQX
M#2!S96,-(&QD>"!T-0T@<V)C(&-O<RQX#2!C;&,-(&QD>"!T-@T@861C(&-O
M<RQX(#MD23TH0T]3*%0X*2U#3U,H5#<I*T-/4RA4-BDM0T]3*%0U*2DO,@T@
M/CX^(&1I=C(-(&-L8PT@;&1X('0S#2!A9&,@<VEN+'@-('-E8PT@;&1X('0T
M#2!S8F,@<VEN+'@-('-T82!D,C$@.V0]*%-)3BA4,RDM4TE.*%0T*2MD22DO
M,@TZ8V%L8V4@<V5C#2!L9'@@=#4-(&QD82!S:6XL> T@;&1X('0V#2!S8F,@
M<VEN+'@-('-E8PT@;&1X('0W#2!S8F,@<VEN+'@-('-E8PT@;&1X('0X#2!S
M8F,@<VEN+'@@.V5)/2A324XH5#4I+5-)3BA4-BDM4TE.*%0W*2U324XH5#@I
M*2\R#2 ^/CX@9&EV,@T@8VQC#2!L9'@@=#,-(&%D8R!C;W,L> T@8VQC#2!L
M9'@@=#0-(&%D8R!C;W,L> T@<W1A(&4R,B [93TH0T]3*%0S*2M#3U,H5#0I
M*V5)*2\R#3IC86QC9B!L9'@@=#D-(&QD82!S:6XL> T@<V5C#2!L9'@@=#$P
M#2!S8F,@<VEN+'@-('-T82!F,C,@.V8]*%-)3BA4.2DM4TE.*%0Q,"DI+S(-
M.F-A;&-G(&QD>"!T-@T@;&1A('-I;BQX#2!S96,-(&QD>"!T. T@<V)C('-I
M;BQX#2!S96,-(&QD>"!T-PT@<V)C('-I;BQX#2!S96,-(&QD>"!T-0T@<V)C
M('-I;BQX(#MG23TH4TE.*%0V*2U324XH5#@I+5-)3BA4-RDM4TE.*%0U*2DO
M,@T@/CX^(&1I=C(-(&-L8PT@;&1X('0T#2!A9&,@8V]S+'@-('-E8PT@;&1X
M('0S#2!S8F,@8V]S+'@-('-T82!G,S$@.V<]*$-/4RA4-"DM0T]3*%0S*2MG
M22DO,@TZ8V%L8V@@8VQC#2!L9'@@=#8-(&QD82!C;W,L> T@;&1X('0W#2!A
M9&,@8V]S+'@-('-E8PT@;&1X('0U#2!S8F,@8V]S+'@-('-E8PT@;&1X('0X
M#2!S8F,@8V]S+'@@.VA)/2A#3U,H5#8I*T-/4RA4-RDM0T]3*%0U*2U#3U,H
M5#@I*2\R#2 ^/CX@9&EV,@T@8VQC#2!L9'@@=#,-(&%D8R!S:6XL> T@8VQC
M#2!L9'@@=#0-(&%D8R!S:6XL> T@<W1A(&@S,B [:#TH4TE.*%0S*2M324XH
M5#0I*VA)*2\R#3IW:&5W(&-L8PT@;&1X('0Y#2!L9&$@8V]S+'@-(&QD>"!T
M,3 -(&%D8R!C;W,L> T@<W1A(&DS,R [:3TH0T]3*%0Y*2M#3U,H5#$P*2DO
M,@T-*BJ@:50G4Z!!3$R@1$]73DA)3$R@1E)/3:!(15)%+@T-9&]W;FAI;&P-
```

```
M*BHJ*J!C3$5!4J!"549&15(-*J!AH$Q)5%1,1:!-04-23PT-<V5T8G5F(&UA
M8R @.W!55*!"549&15)3H%=(15)%H%1(15F@0T%.H$)%H$A54E0-(&QD82 C
M,#  -('-T82!B=69F97(-(&QD82!Z=&5M<" [:$E'2*!"651%#7-T86)U9B!S
M=&$@8G5F9F5R*S$-(#P\/ T-(#X^/B!S971B=68-8VQR9')A=R!L9'@@(S X
M#2!L9&$@(S P#3IF;V]L(&QD>2 C,#  -.F1O<&4@<W1A("AB=69F97(I+'D-
M(&EN>0T@8FYE(#ID;W!E#2!I;F,@8G5F9F5R*S$-(&1E> T@8FYE(#IF;V]L
M#0TJ*BHJH&U9H$=/3T1.15-3H$)55*!I)TV@0:!$3U!%#2IC;')D<F%WH&QD
M8:!G;&]B>&UI;@TJH&QS<J"@.VY%142@5$^@1T54H$E.5$^@5$A%H%))1TA4
MH$-/3%5-3@TJH&)C8Z Z979E;J [95A03$%)3D5$H$E.H$U/4D6@1$5404E,
MH$)%3$]7#2J@;&1YH",D.#  -*J!S='F@8G5F9F5RH#MP4D5354U!0DQ9H%1(
M25.@5TE,3*!"1:!!H$Q)5%1,10TJH&-L8Z"@.TU/4D6@149&24-)14Y4+@TJ
M.F5V96Z@861CH&)U9F9E<BLQ#2J@<W1AH&)U9F9E<BLQ#2J@;&1AH&=L;V)X
M;6%X#2J@<V5C#2J@<V)CH&=L;V)X;6EN#2J@=&%X#2J@:6YX#2J@;&1YH&=L
M;V)Y;6%X#2J@8F5QH#IR97-E= TJ.GEA>:!L9&&@(R0P, TJH&QD>:!G;&]B
M>6UA> TJ.F)L86B@<W1AH"AB=69F97(I+'D-*J!D97D-*J!C<'F@9VQO8GEM
M:6X-*J!B8W.@.F)L86@-*J!L9&&@8G5F9F5R#2J@96]RH",D.# -*J!S=&&@
M8G5F9F5R#2J@8FYEH#IW:&]P964-*J!I;F.@8G5F9F5R*S$-*CIW:&]P966@
M9&5X#2J@8FYEH#IY87D-*CIR97-E=*!L9&&@(S"@.VY%142@5$^@4D53152@
M5$A%4T6@1U594PTJH'-T8:!G;&]B>&UA> TJH'-T8:!G;&]B>6UA> TJH&QD
M8: C)&9F#2J@<W1AH&=L;V)X;6EN#2J@<W1AH&=L;V)Y;6EN#0TJ*BHJH&Y%
M6%0LH%)%042@04Y$H$1205>@4$],64=/3E,-#7)E861D<F%W(&QD>2 C,# -
M('-T>2!I;F1E> UO8FIL;V]P(&QD>2!I;F1E> T@;&1A('!O;'EL:7-T+'D@
M.V9)4E-4+*!42$6@3E5-0D52H$]&H%!/24Y44PT@8FYE(#IC;VYT(#MB552@
M24:@3E5-4$])3E13H$E3H%I%4D^@5$A%3@T@:FUP(&]B:F1O;F4@.U=%H$%2
M1:!!5*!42$6@14Y$H$]&H%1(1:!,25-4#3IC;VYT('-T82!C;W5N='!T<PT@
M:6YC(&EN9&5X#0TJH')/5$%41:!04D]*14-4H$%.1*!$4D%7H%1(1:!03TQ9
M1T].#2J@;4%+1:!355)%H$)51D9%4J!"14E.1Z!$4D%73J!43Z!)4Z!#3$5!
M4B$-#3ID;VET(&IS<B!R;W1P<F]J#0TJH&-/3E9%4E2@6$U)3J!!3D2@6$U!
M6*!43Z!#3TQ534Y3#0T@;&1A(&QO8WAM:6X-(&QS<@T@;'-R#2!L<W(@(#M8
MH$U/1* X#2!S=&$@;&]C>&UI;@T@8VUP(&=L;V)X;6EN#2!B8W,@.FYA: T@
M<W1A(&=L;V)X;6EN#3IN86@@;&1A(&QO8WEM:6X-(&-M<"!G;&]B>6UI;@T@
M8F-S(#IU:'5H#2!S=&$@9VQO8GEM:6X-.G5H=6@@;&1A(&QO8WAM87@-(&QS
M<@T@;'-R#2!L<W(-('-T82!L;V-X;6%X#2!C;7 @9VQO8GAM87@-(&)C8R Z
M;F]W87D-('-T82!G;&]B>&UA> TZ;F]W87D@;&1A(&QO8WEM87@-(&-M<"!G
M;&]B>6UA> T@8F-C(&5O<F9I;&P-('-T82!G;&]B>6UA> T-*J!I1J!54TE.
M1Z!42$6@96]R+4)51D9%4BR@0T]06:!)3E1/H$1205=)3D>@0E5&1D52#2J@
M84Y$H%1(14Z@0TQ%05*@5$A%H&5O<BU"549&15(-#65O<F9I;&P@;&1A(&9I
M;&P-(&)E<2!O8FIL;V]P#0T@/CX^('-E=&)U9@T@;&1A(",\96]R8G5F#2!S
M=&$@=&5M<#$-(&QD82 C/F5O<F)U9@T@<W1A('1E;7 Q*S$-#2!L9&$@;&]C
M>&UI;B [;&]C>&UI;J!.3U>@0T].5$%)3E.@0T],54U.#2!L<W(@(#ME04-(
MH$-/3%5-3J!)4Z Q,CB@0EE415,-(&)C8R Z979E;B [<T^@5$A%4D6@34E'
M2%2@0D6@0:!#05)260T@;&1Y(",D.# -('-T>2!B=69F97(-('-T>2!T96UP
M,0T@8VQC#3IE=F5N('-T82!T,@T@861C(&)U9F9E<BLQ#2!S=&$@8G5F9F5R
M*S$@.V5!0TB@0T],54U.H$E3H#$R.*!"651%4PT@;&1A('0R#2!A9&,@=&5M
M<#$K,2 [;D]7H%=%H%=)3$R@4U1!4E2@052@5$A%#2!S=&$@=&5M<#$K,2 [
M0T],54U.#0T@;&1A(&QO8WAM87@-('-E8PT@<V)C(&QO8WAM:6X-('1A>" [
M=$]404R@3E5-0D52H$]&H$-/3%5-3E.@5$^@1$\-(&EN>" @.T4N1RZ@1DE,
M3*!#3TQ534Y3H#$N+C,-(&QD>2!L;V-Y;6%X#2!B;F4@.F9O;W -(&EN8R!L
M;V-Y;6%X#3IF;V]P(&QD>2!L;V-Y;6%X#2!L9&$@(S P#3IG;V]P(&5O<B H
M=&5M<#$I+'D@.V5O<BU"549&15(-('!H80TJH&U!64)%H%!55*!!3J!E;W*@
M0D5,3U<_#2!E;W(@*&)U9F9E<BDL>0T@<W1A("AB=69F97(I+'D-(&QD82 C
M,# @.VU)1TA4H$%3H%=%3$R@0TQ%05*@252@3D]7#2!S=&$@*'1E;7 Q*2QY
M#2!P;&$-(&1E>0T@8W!Y(&QO8WEM:6X-(&)C<R Z9V]O< T@;&1A(&)U9F9E
M<@T@96]R(",D.# -('-T82!B=69F97(-('-T82!T96UP,0T@8FYE(#IB;V]P
M#2!I;F,@8G5F9F5R*S$-(&EN8R!T96UP,2LQ#3IB;V]P(&1E> T@8FYE(#IF
M;V]P#2!J;7 @;V)J;&]O< T-;V)J9&]N90TJ*BHJH'-705"@0E5&1D524PT-
M<W=A<&)U9B!L9&$@=FUC<V(-(&5O<B C)# R(#MP4D545%F@5%))0TM9+*!%
M2#\-('-T82!V;6-S8@T@;&1A(",D,#@-(&5O<B!Z=&5M<" [6E1%35 ]2$E'
M2*!"651%H$I54U2@1DQ)4%,-('-T82!Z=&5M<" [0D545T5%3J D,S"@04Y$
MH"0S. T-(&IM<"!M86EN(#MA4D]53D2@04Y$H$%23U5.1*!71:!'3RXN+@T-
M('1X=" G9T5%H&)204E.+*!72$%4H$1/H%E/5:!704Y4H%1/H$1/H"<-('1X
M=" G5$].24=(5#\G#0TJ*J!R3U1!5$4LH%!23TI%0U0LH$%.1*!35$]21:!4
M2$6@4$])3E13#2H-*J!T2$E3H%!!4E2@25.@0:!324=.249)0T%.5*!#2$%.
M1T6@4TE.0T4-*J!6,BXP+J"@;D]7H$E4H$E3H$&@0T]-4$Q%5$5,6:!'14Y%
M4D%,H%!/3%E'3TZ@4$Q/5%1%4BX-*J!AH%-%5*!/1J!03TE.5%.@25.@4D5!
M1*!)3BR@4D]4051%1*!!3D2@4%)/2D5#5$5$+*!!3D0-*J!03$]45$5$H$E.
M5$^@5$A%H$1205=)3D>@0E5&1D52H"AE;W*@3U*@3D]234%,*2X-#7)O='!R
M;VH-#2J@8:!.14%4H$U!0U)/#6YE9R!M86,@(#MC2$%.1T6@5$A%H%-)1TZ@
M3T:@0:!45T\G4Z!#3TU03$5-14Y4#2!C;&,-(&QD82!=,2 [3E5-0D52+@T@
M96]R(",D9F8-(&%D8R C)# Q#2 \/#P-#2HM+2TM+2TM+2TM+2TM+2TM+2TM
M+2TM+2TM+2TM+2TM#2J@=$A%4T6@34%#4D]3H%)%4$Q!0T6@5$A%H%!2159)
M3U53H%!23TI%0U1)3TX-*J!354)23U5424Y%+@T-<VUU;'2@;6%CH#MM54Q4
M25!,6:!45T^@4TE'3D5$H#@M0DE4#2 [3E5-0D524SJ@82IY+S8TH"T^H&$-
M('-T8:!Z,PT@8VQCH* [=$A)4Z!-54Q425!,6:!)4Z!&3U*@3D]234%,#2!E
M;W*@(R1F9J [3E5-0D524RR@22Y%+J!8/2TV-"XN-C0-(&%D8Z C)# Q#2!S
M=&&@>C0-(&QD8: H>C,I+'D-('-E8PT@<V)CH"AZ-"DL>0T@/#P\H* [84Q,
MH$1/3D6@.BD-#7-M=6QT>B!M86,@.VU53%1)4$Q9H%173Z!324=.142@."U"
M250-(" @.TY534)%4E,ZH&$J>2\V-* M/J!A#2!S=&$@>C$-(&-L8R @.V%.
M1*!42$E3H$U53%1)4$Q9H$E3H%-014-)1DE#04Q,60T@96]R(",D9F8@.T9/
M4J!42$6@4%)/2D5#5$E/3J!005)4+*!72$5210T@861C(",D,#$@.TY534)%
M4E.@05)%H"TQ,3 N+C$Q,*!!3D2@,"XN-# -('-T82!Z,@T@;&1A("AZ,2DL
```

```
M>0T@<V5C#2!S8F,@*'HR*2QY#2 \/#P@(#MA3$R@1$].1: Z*0T-<')O:F5C
M="!M86,@(#MT2$6@04-454%,H%!23TI%0U1)3TZ@4D]55$E.10T[=$A%H%)/
M551)3D6@5$%+15.@5$A%H%!/24Y4#3M=,:!=,J!=,RR@4D]4051%4Z!!3D0-
M.U!23TI%0U13H$E4+*!!3D2@4U1/4D53H%1(10T[4D5354Q4H$E.H%TQH%TR
MH%TS+@T-(&QD>2!=,2 [;55,5$E03%F@1DE24U2@4D]4051)3TZ@0T],54U.
M#2!L9&$@83$Q#2 ^/CX@<VUU;'0-('-T82!P,70-(&QD82!D,C$-(#X^/B!S
M;75L= T@<W1A(' R= T@;&1A(&<S,0T@/CX^('-M=6QT#2!S=&$@<#-T#2!L
M9'D@73(@.W-%0T].1*!#3TQ534X-(&QD82!B,3(-(#X^/B!S;75L= T@8VQC
M#2!A9&,@<#%T#2!S=&$@<#%T#2!L9&$@93(R#2 ^/CX@<VUU;'0-(&-L8PT@
M861C(' R= T@<W1A(' R= T@;&1A(&@S,@T@/CX^('-M=6QT#2!C;&,-(&%D
M8R!P,W0-('-T82!P,W0-(&QD>2!=,R [=$A)4D2@0T],54U.#2!L9&$@8S$S
M#2 ^/CX@<VUU;'0-(&-L8PT@861C(' Q= T@<W1A(' Q= T@;&1A(&8R,PT@
M/CX^('-M=6QT#2!C;&,-(&%D8R!P,G0-('-T82!P,G0-(&QD82!I,S,-(#X^
M/B!S;75L= T@8VQC#2!A9&,@<#-T#2!S=&$@73,@.W)/5$%4142@>@T@=&%X
M#2!L9'D@>F1I=BQX(#MT04),1:!/1J!$+RA:*UHP*0T@(" [;D]7H'F@0T].
M5$%)3E.@4%)/2D5#5$E/3J!#3TY35 T-(&QD82!P,70-(#X^/B!S;75L='H-
M(&QD>"!Z;V]M#2!C<'@@(S8T#2!B97$@8V]N='@-('-T>2!T96UP,0T@;&1Y
M('IO;VT-(#X^/B!S;75L= T@;&1Y('1E;7 Q#6-O;G1X(&-L8PT@861C(",V
M-" [;T9&4T54H%1(1:!#3T]21$E.051%#2!S=&$@73$@.W)/5$%4142@04Y$
MH%!23TI%0U1%1 T@8VUP(&QO8WAM:6X@.W-%1:!)1J!)5*!)4Z!!H$Q/0T%,
MH$U)3DE-54T-(&)C<R!N;W1X;6EN#2!S=&$@;&]C>&UI;@UN;W1X;6EN(&-M
M<"!L;V-X;6%X#2!B8V,@;F]T>&UA> T@<W1A(&QO8WAM87@-#6YO='AM87@@
M;&1A(' R= T@/CX^('-M=6QT>@T@8W!X(",V- T@8F5Q(&-O;G1Y#2!L9'D@
M>F]O;0T@/CX^('-M=6QT#6-O;G1Y(&-L8PT@861C(",V- T@<W1A(%TR(#MR
M3U1!5$5$H$%.1*!04D]*14-4142@>0T@8VUP(&QO8WEM:6X-(&)C<R!N;W1Y
M;6EN#2!S=&$@;&]C>6UI;@UN;W1Y;6EN(&-M<"!L;V-Y;6%X#2!B8V,@;F]T
M>6UA> T@<W1A(&QO8WEM87@-#6YO='EM87@@/#P\(" [84Q,H$1/3D4-#2J@
M;&1AH",\96]R8G5FH#MF25)35*!71:!.145$H%1/H$-,14%2H%1(10TJH'-T
M8:!B=69F97*@.V5O<J!"549&15(-*J!L9&&@(SYE;W)B=68-*J!S=&&@8G5F
M9F5R*S$-#2!L9&$@(S @.W)%4T54H'E-24Z@04Y$H'E-05@-('-T82!L;V-Y
M;6%X#2!S=&$@;&]C>&UA> T@;&1A(",D9F8-('-T82!L;V-Y;6EN#2!S=&$@
M;&]C>&UI;@T-<F5A9'!T<R!L9'D@:6YD97@-(&QD82!P;VQY;&ES="QY#2!S
M=&$@<#%X#2!I;GD-(&QD82!P;VQY;&ES="QY#2!S=&$@<#%Y#2!I;GD-(&QD
M82!P;VQY;&ES="QY#2!S=&$@<#%Z#2!I;GD-(&1E8R!C;W5N='!T<PT@;&1A
M('!O;'EL:7-T+'D-('-T82!P,G@-(&EN>0T@;&1A('!O;'EL:7-T+'D-('-T
M82!P,GD-(&EN>0T@;&1A('!O;'EL:7-T+'D-('-T82!P,GH-(&EN>0T@9&5C
M(&-O=6YT<'1S#2!L9&$@<&]L>6QI<W0L>0T@<W1A(' S> T@:6YY#2!L9&$@
M<&]L>6QI<W0L>0T@<W1A(' S>0T@:6YY#2!L9&$@<&]L>6QI<W0L>0T@<W1A
M(' S>@T@:6YY#2!S='D@:6YD97@-(#X^/B!P<F]J96-T+' Q>#MP,7D[<#%Z
M#2 ^/CX@<')O:F5C="QP,G@[<#)Y.W R>@T@/CX^('!R;VIE8W0L<#-X.W S
M>3MP,WH-#2!L9&$@:&ED90T@8F5Q(#ID;VET#2!L9&$@<#)X(#MH241$14Z@
M1D%#1:!#2$5#2PT@<V5C#2!S8F,@<#%X#2!T87D@(#MY/2A8,BU8,2D-(&QD
M82!P,WD-('-E8PT@<V)C(' R>2 [83TH63,M63(I#2 ^/CX@<VUU;'0-('-T
M82!T96UP,0T@;&1A(' S> T@<V5C#2!S8F,@<#)X#2!T87D-(&QD82!P,GD-
M('-E8PT@<V)C(' Q>0T@/CX^('-M=6QT#2!C;7 @=&5M<#$@.VE&H%@Q*EDR
M+5DQ*E@RH#Z@,*!42$5.H$9!0T4-(&)M:2 Z9&]I=" [25.@5DE324),10T@
M9&5C(&-O=6YT<'1S(#MO5$A%4E=)4T6@4D5!1*!)3J!214U!24Y)3D<-(&)E
M<2 Z86)O<G0@.U!/24Y44Z!!3D2@4D5455).#3IP;V]P(&EN8R!I;F1E> T@
M:6YC(&EN9&5X#2!I;F,@:6YD97@-(&1E8R!C;W5N='!T<PT@8FYE(#IP;V]P
M#3IA8F]R="!R=',-#3ID;VET(&QD82!P,7@-('-T82!X,0T@;&1A(' Q>0T@
M<W1A('DQ#2!L9&$@<#)X#2!S=&$@>#(-(&QD82!P,GD-('-T82!Y,@T@:G-R
M(&1R87<-(&QD82!P,G@-('-T82!X,0T@;&1A(' R>0T@<W1A('DQ#2!L9&$@
M<#-X#2!S=&$@>#(-(&QD82!P,WD-('-T82!Y,@T@:G-R(&1R87<-#2!D96,@
M8V]U;G1P=',-(&)N92!P;VQY;&]O<" [:5.@252@2E535*!!H%1224%.1TQ%
M/PT@:FUP('!O;'ED;VYE#0UP;VQY;&]O<"!L9'D@:6YD97@-(&QD82!P;VQY
M;&ES="QY#2!S=&$@<#)X#2!I;GD-(&QD82!P;VQY;&ES="QY#2!S=&$@<#)Y
M#2!I;GD-(&QD82!P;VQY;&ES="QY#2!S=&$@<#)Z#2!I;GD-('-T>2!I;F1E
M> T@/CX^('!R;VIE8W0L<#)X.W R>3MP,GH-#2!L9&$@<#)X#2!S=&$@>#$-
M(&QD82!P,GD-('-T82!Y,0T@;&1A(' S> T@<W1A('@R#2!L9&$@<#-Y#2!S
M=&$@>3(-(&IS<B!D<F%W#0T@;&1A(' R> T@<W1A(' S> T@;&1A(' R>0T@
M<W1A(' S>0T@9&5C(&-O=6YT<'1S#2!B97$@<&]L>61O;F4-(&IM<"!P;VQY
M;&]O< UP;VQY9&]N92!L9&$@<#%X(#MC3$]31:!42$6@4$],64=/3@T@<W1A
M('@R#2!L9&$@<#%Y#2!S=&$@>3(-(&QD82!P,W@-('-T82!X,0T@;&1A(' S
M>0T@<W1A('DQ#2!J<W(@9')A=PT@<G1S#0T@='AT("=S04U%H%1(24Y'H%=%
MH$1/H$5615)9H$Y)1TA4+*!P24Y+63J@)PT@='AT("=44EF@5$^@5$%+1:!/
M5D52H%1(1:!73U),1"$G#0T-*BTM+2TM+2TM+2TM+2TM+2TM+2TM+2TM+2TM
M+2TM+2T-*J!G14Y%4D%,H%%515-424].04),12U604Q51:!%4E)/4J!04D]#
M14154D4-#2IC:&]K9:!L9'B@(S P#2HZ;&]O<*!L9&&@.F-T97AT+'@-*J!B
M97&@.F1O;F4-*J!J<W*@8VAR;W5T#2J@:6YX#2J@:FUPH#IL;V]P#2HZ9&]N
M9:!R=',-*CIC=&5X=*!H97B@,&2@.V-R#2J@='ATH"=33TU%5$A)3D>@0TA/
M2T5$H#HH)PTJH&AE>* P9# P#2H-('1X=" G;D%21B$G#0TJ+2TM+2TM+2TM
M+2TM+2TM+2TM+2TM+2TM+2TM+0TJH&1205=)3B>@0:!,24Y%+J"@8:!&
M04A.H$Q!2$XN#0TJ*BJ@<T]-1:!54T5&54R@34%#4D]3#0UC:6YI="!M86,@
M(#MM04-23Z!43Z!)3DE424%,25I%H%1(1:!#3U5.5$52#2!L9&$@73$@.T18
MH$]2H$19#2!L<W(-(#P\/" @.W1(1:!$6"\RH$U!2T53H$&@3DE#15*@3$]/
M2TE.1Z!,24Y%#0TJ*BHJ*J!M04-23Z!43Z!404M%H$&@4U1%4*!)3J!X#0UX
M<W1E<"!M86,-(&QD>"!D>" [;E5-0D52H$]&H$Q/3U"@251%4D%424].4PT@
M/CX^(&-I;FET+&1X#7AL;V]P(&QS<B!C:'5N:PT@8F5Q(&9I>&,@.W501$%4
M1:!#3TQ534X-('-B8R!D>0T@8F-C(&9I>'D@.W1)346@5$^@4U1%4*!)3J!Y
M#2!D97@-(&)N92!X;&]O< UD;VYE(&QD82!O;&1X(#MP3$]4H%1(1:!,05-4
MH$-(54Y+#2!E;W(@8VAU;FL-(&]R82 H8G5F9F5R*2QY#2!S=&$@*&)U9F9E
```

```
M<BDL>0T@<G1S#0UF:7AC('!H80T@;&1A(&]L9'@-(&]R82 H8G5F9F5R*2QY
M(#MP3$]4#2!S=&$@*&)U9F9E<BDL>0T@;&1A(",D9F8@.W501$%41:!#2%5.
M2PT@<W1A(&]L9'@-('-T82!C:'5N:PT@;&1A(",D.# @.VE.0U)%05-%H%1(
M1:!#3TQ534X-(&5O<B!B=69F97(-('-T82!B=69F97(-(&)N92!C,@T@:6YC
M(&)U9F9E<BLQ#6,R#2!P;&$-('-B8R!D>0T@8F-S(&-O;G0-(&%D8R!D> T@
M:68@:2Q=,2 [9$^@5T6@55-%H&EN>:!/4J!D97D_#2!I;GD-(&5L<V4-(&1E
M>0T@9FEN#6-O;G0@9&5X#2!B;F4@>&QO;W -(&IM<"!D;VYE#0UF:7AY(&%D
M8R!D> T@<&AA#2!L9&$@;VQD> T@96]R(&-H=6YK#2!O<F$@*&)U9F9E<BDL
M>0T@<W1A("AB=69F97(I+'D-(&QD82!C:'5N:PT@<W1A(&]L9'@-('!L80T@
M:68@:2Q=,2 [=5!$051%H'D-(&EN>0T@96QS90T@9&5Y#2!F:6X-(&1E> T@
M8FYE('AL;V]P#2!R=',-(#P\/" @.V5.1*!/1J!M04-23Z!84U1%4 T-*BHJ
M*BJ@=$%+1:!!H%-415"@24Z@>0T->7-T97 @;6%C#2!L9'@@9'D@.VY534)%
M4J!/1J!,3T]0H$E415)!5$E/3E,-(&)E<2!D;VYE(#MI1J!$63TPH$E4)U.@
M2E535*!!H%!/24Y4#2 ^/CX@8VEN:70L9'D-('-E8PUY;&]O<"!P:&$-(&QD
M82!O;&1X#2!O<F$@*&)U9F9E<BDL>0T@<W1A("AB=69F97(I+'D-('!L80T@
M:68@:2Q=,0T@:6YY#2!E;'-E#2!D97D-(&9I;@T@<V)C(&1X#2!B8V,@9FEX
M> T@9&5X#2!B;F4@>6QO;W -9&]N92!L9&$@;VQD> T@;W)A("AB=69F97(I
M+'D-('-T82 H8G5F9F5R*2QY#2!R=',-#69I>'@@861C(&1Y#2!L<W(@;VQD
M> T@<V5C(" [:4U03U)404Y4(0T@8F5Q(&9I>&,-(&1E> T@8FYE('EL;V]P
M#2!J;7 @9&]N90T-9FEX8R!P:&$-(&QD82 C)#@P#2!S=&$@;VQD> T@96]R
M(&)U9F9E<@T@<W1A(&)U9F9E<@T@8FYE(&,R#2!I;F,@8G5F9F5R*S$-8S(@
M<&QA#2!D97@-(&)N92!Y;&]O< T@:FUP(&1O;F4-(#P\/" @.V5.1*!/1J!M
M04-23Z!94U1%4 T-*J!T04M%H$%.H%B@4U1%4*!)3J!42$6@96]RH$)51D9%
M4@TJH'1(1:!33TQ%H$-(04Y'1:!)4Z!43Z!54T6@96]RH$E.4U1%042@3T:@
M;W)A#0UE;W)X<W1E<"!M86,-(&QD>"!D>" [;E5-0D52H$]&H$Q/3U"@251%
M4D%424].4PT@/CX^(&-I;FET+&1X#7AL;V]P(&QS<B!C:'5N:PT@8F5Q(&9I
M>&,@.W501$%41:!#3TQ534X-('-B8R!D>0T@8F-C(&9I>'D@.W1)346@5$^@
M4U1%4*!)3J!Y#2!D97@-(&)N92!X;&]O< UD;VYE(&QD82!O;&1X(#MP3$]4
MH%1(1:!,05-4H$-(54Y+#2!E;W(@8VAU;FL-(&5O<B H8G5F9F5R*2QY#2!S
M=&$@*&)U9F9E<BDL>0T@<G1S#0UF:7AC('!H80T@;&1A(&]L9'@-(&5O<B H
M8G5F9F5R*2QY(#MP3$]4#2!S=&$@*&)U9F9E<BDL>0T@;&1A(",D9F8@.W50
M1$%41:!#2%5.2PT@<W1A(&]L9'@-('-T82!C:'5N:PT@;&1A(",D.# @.VE.
M0U)%05-%H%1(1:!#3TQ534X-(&5O<B!B=69F97(-('-T82!B=69F97(-(&)N
M92!C,@T@:6YC(&)U9F9E<BLQ#6,R#2!P;&$-('-B8R!D>0T@8F-S(&-O;G0-
M(&%D8R!D> T@:68@:2Q=,2 [9$^@5T6@55-%H&EN>:!/4J!D97D_#2!I;GD-
M(&5L<V4-(&1E>0T@9FEN#6-O;G0@9&5X#2!B;F4@>&QO;W -(&IM<"!D;VYE
M#0UF:7AY(&%D8R!D> T@<&AA#2!L9&$@;VQD> T@96]R(&-H=6YK#2!E;W(@
M*&)U9F9E<BDL>0T@<W1A("AB=69F97(I+'D-(&QD82!C:'5N:PT@<W1A(&]L
M9'@-('!L80T@:68@:2Q=,2 [=5!$051%H'D-(&EN>0T@96QS90T@9&5Y#2!F
M:6X-(&1E> T@8FYE('AL;V]P#2!R=',-(#P\/" @.V5.1*!/1J!M04-23Z!8
M4U1%4 T-#2J@=$%+1:!!H%DM4U1%4*!)3J!42$6@96]R+4)51D9%4@TJH&-(
M04Y'15.@1E)/3:!!0D]61:!!4D4ZH$].3%F@4$Q/5*!,05-4H%!!4E2@3T:@
M14%#2 TJH%9%4E1)0T%,H$-(54Y++*!$3TXG5*!03$]4H$Q!4U2@4$])3E0L
MH%!,3U2@5TE42*!E;W(-#65O<GES=&5P(&UA8PT@;&1X(&1Y(#MN54U"15*@
M3T:@3$]/4*!)5$52051)3TY3#2!B97$@9&]N92 [:4:@1%D],*!)5"=3H$I5
M4U2@0:!03TE.5 T@/CX^(&-I;FET+&1Y#2!S96,-*GEL;V]PH'!H80TJH&QD
M8:!O;&1X#2J@;W)AH"AB=69F97(I+'D-*J!S=&&@*&)U9F9E<BDL>0TJH'!L
M80UY;&]O<"!I9B!I+%TQ#2!I;GD-(&5L<V4-(&1E>0T@9FEN#2!S8F,@9'@-
M(&)C8R!F:7AX#2!D97@-(&)N92!Y;&]O< TJ9&]N9:!L9&&@;VQD> TJH&]R
M8: H8G5F9F5R*2QY#2J@<W1AH"AB=69F97(I+'D-9&]N92!R=',-#69I>'@@
M861C(&1Y#2!P:&$@(#MW1:!/3DQ9H%!,3U2@5$A%H$Q!4U2@4$%25*!/1J!%
M04-(H$-(54Y+#2!L9&$@;VQD> T@96]R("AB=69F97(I+'D-('-T82 H8G5F
M9F5R*2QY#2!P;&$-(&QS<B!O;&1X#2!S96,@(#MI35!/4E1!3E0A#2!B97$@
M9FEX8PT@9&5X#2!B;F4@>6QO;W -(&IM<"!D;VYE#0UF:7AC('!H80T@;&1A
M(",D.# -('-T82!O;&1X#2!E;W(@8G5F9F5R#2!S=&$@8G5F9F5R#2!B;F4@
M8S(-(&EN8R!B=69F97(K,0UC,B!P;&$-(&1E> T@8FYE('EL;V]P#2!J;7 @
M9&]N90T@/#P\" [94Y$H$]&H&U!0U)/H%E35$50#2HJ*BJ@:4Y)5$E!3*!,
M24Y%H%-%5%50#0TJ*J!T2$6@0T]-345.5$5$H$Q)3D53H$)%3$]7H$%21:!.
M3U>@5$%+14Z@0T%21:!/1J!"6:!42$4-*BJ@0T%,3$E.1Z!23U5424Y%+@TJ
M9')A=Z ^/CZ@;6]V92QT>#$[>#&@H#MM3U9%H%-4549&H$E.5$^@6D523Z!0
M04=%#2J@/CX^H&UO=F4L='@R.W@RH* [=TA%4D6@252@0T%.H$)%H$U/1$E&
M245$#2J@/CX^H&UO=F4L='DQ.WDQ#2J@/CX^H&UO=F4L='DR.WDR#0UD<F%W
M(&QD82!F:6QL#2!B;F4@.G-E=&5O<@T@/CX^('-E=&)U9@T@:FUP(#IS971U
M< TZ<V5T96]R(&QD82 C/&5O<F)U9B [=5-%H&5O<J!"549&15*@24Y35$5!
M1*!/1@T@<W1A(&)U9F9E<B [1$E34$Q!6:!"549&15*@1D]2H$1205=)3D<-
M(&QD82 C/F5O<F)U9@T@<W1A(&)U9F9E<BLQ#0TZ<V5T=7 @<V5C(" [;4%+
M1:!355)%H%@Q/%@R#2!L9&$@>#(-('-B8R!X,0T@8F-S(#IC;VYT#2!L9&$@
M>3(@.VE&H$Y/5"R@4U=!4*!P,:!!3D2@<#(-(&QD>2!Y,0T@<W1A('DQ#2!S
M='D@>3(-(&QD82!X,0T@;&1Y('@R#2!S='D@>#$-('-T82!X,@T-('-E8PT@
M<V)C('@Q(#MN3U>@83U$6 TZ8V]N="!S=&$@9'@-(&QD>"!X,2 [<%54H%@Q
MMH$E.5$^@>"R@3D]7H%=%H$-!3J!44D%32*!X,0T-8V]L=6UN('1X82 [9DE.
M1*!42$6@1DE24U2@0T],54U.H$9/4J!X#2!L<W(-(&QS<B @.W1(15)%H$%2
M1:!8,2\XH#$R.*!"651%H$),3T-+4PT@;'-R(" [=TA)0TB@345!3E.@6#$O
M,3:@,C4VH$)95$6@0DQ/0TM3#2!L<W(-(&)C8R Z979E;B [=TE42*!!H%!/
M4U-)0DQ%H$585%)!H#$R.*!"651%H$),3T-+#2!L9'D@(R0X," [24:@4T\L
MH%-%5*!42$6@2$E'2*!"250-('-T>2!B=69F97(-(&-L8PTZ979E;B!A9&,@
M8G5F9F5R*S$@.V%$1*!)3J!42$6@3E5-0D52H$]&H#(U-J!"651%H$),3T-+
M4PT@<W1A(&)U9F9E<BLQ#0T@<V5C#2!L9&$@>3(@.V-!3$-53$%41:!$60T@
M<V)C('DQ#2!B8W,@.F-O;G0R(#MI4Z!9,CY9,3\-(&5O<B C)&9F(#MO5$A%
M4E=)4T6@1%D]63$M63(-(&%D8R C)# Q#3IC;VYT,B!S=&$@9'D-(&-M<"!D
M>" [=TA/)U.@0DE'1T52.J!$6:!/4J!$6#\-(&)C8R!S=&5P:6YX(#MI1J!$
```

```
M6"R@5$A%3BXN+@T@:FUP('-T97!I;GD-#7-T97!I;G@@;&1Y('DQ#2!C<'D@
M>3(-(&QD82!B:71P+'@@.WB@0U524D5.5$Q9H$-/3E1!24Y3H%@Q#2!S=&$@
M;VQD> T@<W1A(&-H=6YK#2!B8V,@>&EN8WD@.V1/H%=%H%-415"@1D]25T%2
M1%.@3U*@0D%#2U=!4D13H$E.H'D_#2!J;7 @>&1E8WD-#7AI;F-Y(&QD82!F
M:6QL#2!B97$@;F]R;7AI;F,-(#X^/B!E;W)X<W1E<"QI;GD-;F]R;7AI;F,@
M/CX^('AS=&5P+&EN>0T->&1E8WD@;&1A(&9I;&P-(&)E<2!N;W)M>&1E8PT@
M/CX^(&5O<GAS=&5P+&1E>0UN;W)M>&1E8R ^/CX@>'-T97 L9&5Y#0US=&5P
M:6YY(&QD>2!Y,0T@;&1A(&)I=' L>" [>#U8,0T@<W1A(&]L9'@-(&QS<B @
M.WF@1$]%4TXG5*!54T6@0TA53DM3#2!E;W(@;VQD>" [<T^@5T6@2E535*!7
M04Y4H%1(1:!"250-('-T82!O;&1X#2!C<'D@>3(-(&)C<R!Y9&5C>0T->6EN
M8WD@;&1A(&9I;&P-(&)E<2!N;W)M:6YC#2 ^/CX@96]R>7-T97 L:6YY#6YO
M<FUI;F,@/CX^('ES=&5P+&EN>0T->61E8WD@;&1A(&9I;&P-(&)E<2!N;W)M
M9&5C#2 ^/CX@96]R>7-T97 L9&5Y#6YO<FUD96,@/CX^('ES=&5P+&1E>0T-
M#2HM+2TM+2TM+2TM+2TM+2TM+2TM+2TM+2TM+2TM+2TM#2J@8TQ%04Z@55 -
M#6-L96%N=7 @;&1A('9M8W-B(#MS5TE40TB@0TA!4J!23TV@0D%#2Z!)3@T@
M86YD(",E,3$Q,3 Q,#$@.T1%1D%53%0-('-T82!V;6-S8@T-(')T<R @.T)9
M12$-#2!T>'0@)U-024Y!3*!#4D%#2T52H"<-('1X=" G4TQ*H#8O.34G#0TJ
M+2TM+2TM+2TM+2TM+2TM+2TM+2TM+2TM+2TM+2TM+0TJH'-%5*!54*!"252@
M5$%"3$4-#2!D<R!>(#MC3$5!4J!43Z!%3D2@3T:@4$%'10T@(" [<T^@5$A!
M5*!404),15.@4U1!4E2@3TZ@0:!004=%H$)/54Y$05)9#6)I=' @;'5P(#$V
M(#LQ,CB@94Y44DE%4Z!&3U*@> T@9&9B("4Q,3$Q,3$Q,0T@9&9B("4P,3$Q
M,3$Q,0T@9&9B("4P,#$Q,3$Q,0T@9&9B("4P,# Q,3$Q,0T@9&9B("4P,# P
M,3$Q,0T@9&9B("4P,# P,#$Q,0T@9&9B("4P,# P,# Q,0T@9&9B("4P,# P
M,# P,0T@+2U>#0US:6X@.W1!0DQ%H$]&H%-)3D53+* Q,C"@0EE415,-8V]S
M(&5Q=2!S:6XK,3(X(#MT04),1:!/1J!#3U-)3D53#2 @(#MB3U1(H$]&H%1(
M15-%H%1224>@5$%"3$53H$%210T@(" [0U524D5.5$Q9H%-%5*!54*!&4D]-
MH&)A<VEC#7ID:78@97%U(&-O<RLQ,C@@.V1)5DE324].H%1!0DQ%#71M871H
M,2!E<74@>F1I=BLS.#0@.VU!5$B@5$%"3$6@3T:@1BA8*3U8*E@O,C4V#71M
M871H,B!E<74@=&UA=&@Q*S4Q,B [<T5#3TY$H$U!5$B@5$%"3$4-<&]L>6QI
M<W0@97%U('1M871H,BLU,3(@.VQ)4U2@3T:@4$],64=/3E,-&AH:&AH:&AH:
8&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:

end

begin 666 cube3d3.2.o
M ("I (T@T(TAT*T8T"D/"1"-&-"@ *D?A?NI@(7\3+R!DP41$1$@(" @(" @
M(" @(" @0U5"13-$(%8S+C(-#2 @(" @(" @(" @(" @(" @($)9#9\@(" @
M4U1%4$A%3B!*541$F2 @("!'14]21T4@5$%93$]2#0V;("!#2$5#2R!/550@
M5$A%($I!3BX@.34@25-3544@3T8-EB @0SU(04-+24Y'FR!&3U(@34]212!$
M151!24Q3(0T-'1V>$D8Q+T8RDB M($E.0R]$14,@6"U23U1!5$E/3@T='1)&
M,R]&-)(@+2!)3D,O1$5#(%DM4D]4051)3TX-'1T21C4O1C:2("T@24Y#+T1%
M0R!:+5)/5$%424].#1T=$B!&-R @DB M(%)%4T54#1T=$B K+RT@DB M(%I/
M3TT@24XO3U54#1T=$B @2" @DB M(%1/1T=,12!(241$14X@4U521D%#15,-
M'1T24U!!0T62("T@5$]'1TQ%(%-54D9!0T4@1DE,3$E.1PT-("!04D534R!1
M(%1/(%%5250-#04@(" @("!04D534R!!3ED@2T59(%1/($)%1TE.#0"Q^_ +
M(-+_R-#VYOQ,O($@Y/_) /#YJ9*%(X4EJ92%)X4IJ0&-(="IDR#2_ZD C2'0
MJ4!I#(7[J06%_*D H "B !B1^\AI$)#Y&*7[:2B%^Z7\:0"%_*  Z(K@$-#D
MJ0"%HZDPA:2@ *(8J0"1H\C0^^:DRM#VJ0"%HZDPA:2% JT8T"GQ"0Z-&-"I
M (57A5B%685@A3^%085 A4*%885BA6.%9(5EA6:%4*D!A;6I0(5Q6"#D_\F%
MT NE8<D\\%'F84P-@\F)T FE8?!$QF%,#8/)AM +I6+)// UYF),#8/)BM )
MI6+P*,9B3 V#R8?0"Z5CR3SP&>9C3 V#R8O0":5C\ S&8TP-@\F(T 9,1H),
M#8/)*] 'YG'F<4P-@\DMT W&<<9Q$"CF<>9Q3 V#R4C0":6U20&%M4P-@\D@
MT FE4$D!A5!,#8/)4= #3)F.>!BE9&5AR7B0 NEXA608I65E8LEXD +I>(5E
M&*5F96/)>) "Z7B%9CBE9>5FL )I>(5G&*5E96;)>) "Z7B%:!BE9&5FR7B0
M NEXA6DXI63E9K ":7B%:ABE9&5HR7B0 NEXA6LXI63E9[ ":7B%;!BE9&5G
MR7B0 NEXA6TXI6CE9+ ":7B%;CBE9>5DL )I>(5O&*5D967)>) "Z7B%<!BF
M9[T D*9H?0"0A:6F9[V CSBF:/V CX6FIF6]@(\0#AA)_VD!"AA)_VD!3-V#
M"H6G.*9NO0"0IFW] ) XIFO] ) 8IFQ] ) 0#AA)_VD!2AA)_VD!3 >$2ABF
M:7V CSBF:OV CX6H.*9KO8"/IFS]@(\XIFW]@(\XIF[]@(\0#AA)_VD!2AA)
M_VD!3#V$2ABF:7T D!BF:GT D(6IIF^]@(\XIG#]@(^%JJ9LO8"/.*9N_8"/
M.*9M_8"/.*9K_8"/$ X82?]I 4H82?]I 4R A$H8IFI] ) XIFG] )"%JQBF
M;+T D*9M?0"0.*9K_0"0.*9N_0"0$ X82?]I 4H82?]I 4RVA$H8IFE]@(\8
MIFI]@(^%K!BF;[T D*9P?0"0A:VI (6CI0*%I*((J0"@ )&CR-#[YJ3*T/2@
M (11I%&Y );0 TR*A852YE$@Q86E5TI*2H57Q3^P H4_I5G%0; "A4&E6$I*
M2H58Q4"0 H5 I6#%0I "A4*E4/#!J0"%HZ4"A:2I (7[J4"%_*572I 'H("$
MHX3[&(5H9:2%I*5H9?R%_*58..57JNBD8- "YF"D8*D 4?M(4:.1HZD D?MH
MB,19L.^EHTF A:.%^] $YJ3F_,K0VDSMA*T8T$D"C1C0J0A% H4"3&Z"9T5%
M(&)204E.+"!72$%4($1/(%E/52!704Y4(%1/($1/(%1/3DE'2%0_J0"%8(58
MJ?^%6857I%&Y ):%DLBY ):%D\BY ):%E,C&4KD EH65R+D EH66R+D EH6N
MR,92N0"6A:_(N0"6A;#(N0"6A;'(A%&DDJ6EA2882?]I 84HL28X\2B%LJ6H
MA2882?]I 84HL28X\2B%LZ6KA2882?]I 84HL28X\2B%M*23I::%)AA)_VD!
MA2BQ)CCQ*!AELH6RI:F%)AA)_VD!A2BQ)CCQ*!AELX6SI:R%)AA)_VD!A2BQ
M)CCQ*!AEM(6TI)2EIX4F&$G_:0&%*+$F./$H&&6RA;*EJH4F&$G_:0&%*+$F
M./$H&&6SA;.EK84F&$G_:0&%*+$F./$H&&6TA92JO("0I;*%(AA)_VD!A22Q
M(CCQ)*9QX$#P%(3[I'&%)AA)_VD!A2BQ)CCQ**3[&&E A9+%5[ "A5?%6) "
MA5BELX4B&$G_:0&%)+$B./$DX$#P$*1QA2882?]I 84HL28X\2@8:4"%D\59
ML *%6<5@D *%8*25I:6%)AA)_VD!A2BQ)CCQ*(6RI:B%)AA)_VD!A2BQ)CCQ
M*(6SI:N%)AA)_VD!A2BQ)CCQ*(6TI):EIH4F&$G_:0&%*+$F./$H&&6RA;*E
MJ84F&$G_:0&%*+$F./$H&&6SA;.EK(4F&$G_:0&%*+$F./$H&&6TA;2DKJ6G
MA2882?]I 84HL28X\2@89;*%LJ6JA2882?]I 84HL28X\2@89;.%LZ6MA288
M2?]I 84HL28X\2@89;2%KJJ\@)"ELH4B&$G :0&%)+$B./$DIG'@0/ 4A/ND
```

```
M<84F&$G_:0&%*+$F./$HI/L8:4"%E<57L *%5\58D *%6*6SA2(82?]I 84D
ML2(X\23@0/ 0I'&%)AA)_VD!A2BQ)CCQ*!AI0(66Q5FP H59Q6"0 H5@I*^E
MI84F&$G_:0&%*+$F./$HA;*EJ(4F&$G_:0&%*+$F./$HA;.EJX4F&$G_:0&%
M*+$F./$HA;2DL*6FA2882?]I 84HL28X\2@89;*%LJ6IA2882?]I 84HL28X
M\2@89;.%LZ6LA2882?]I 84HL28X\2@89;2%M*2QI:>%)AA)_VD!A2BQ)CCQ
M*!AELH6RI:J%)AA)_VD!A2BQ)CCQ*!AELX6SI:V%)AA)_VD!A2BQ)CCQ*!AE
MM(6QJKR D*6RA2(82?]I 84DL2(X\22F<>! \!2$^Z1QA2882?]I 84HL28X
M\2BD^QAI0(6OQ5>P H57Q5B0 H58I;.%(AA)_VD!A22Q(CCQ).! \!"D<84F
M&$G_:0&%*+$F./$H&&E A;#%6; "A5G%8) "A6"EM?!'I94XY9*HI; XY9:%
M)AA)_VD!A2BQ)CCQ*(7[I:\XY96HI98XY9.%)AA)_VD!A2BQ)CCQ*,7[, _&
M4O *YE'F4>91QE+0]F"EDH7[I9.%_*65A?VEEH7^(-"+I96%^Z66A?REKX7]
MI;"%_B#0B\92T -,=XND4;D EH65R+D EH66R+D EH6NR(11I)6EI84F&$G_
M:0&%*+$F./$HA;*EJ(4F&$G_:0&%*+$F./$HA;.EJX4F&$G_:0&%*+$F./$H
MA;2DEJ6FA2882?]I 84HL28X\2@89;*%LJ6IA2882?]I 84HL28X\2@89;.%
MLZ6LA2882?]I 84HL28X\2@89;2%M*2NI:>%)AA)_VD!A2BQ)CCQ*!AELH6R
MI:J%)AA)_VD!A2BQ)CCQ*!AELX6SI:V%)AA)_VD!A2BQ)CCQ*!AEM(6NJKR
MD*6RA2(82?]I 84DL2(X\22F<>! \!2$^Z1QA2882?]I 84HL28X\2BD^QAI
M0(65Q5>P H57Q5B0 H58I;.%(AA)_VD!A22Q(CCQ).! \!"D<84F&$G_:0&%
M*+$F./$H&&E A9;%6; "A5G%8) "A6"EE87[I9:%_*6OA?VEL(7^(-"+I96%
MKZ66A;#&4O #3!&*I9*%_:63A?ZEKX7[I;"%_"#0BV!S04U%(%1(24Y'(%=%
M($1/($5615)9($Y)1TA4+"!P24Y+63H@5%)9(%1/(%1!2T4@3U9%4B!42$4@
M5T]23$0A;D%21B&E4- +J0"%HZ4"A:1,YXNI (6CJ4"%I#BE_>7[L!.E_J3\
MA?R$_J7[I/V$^X7]..7[A6>F^XI*2DI*D 6@@(2C&&6DA:0XI?[E_+ $2?]I
M 85HQ6>0 TR-C:3\Q/Z] (^%_87^D -,XXRE4/!3IF>E9TI&_O 0Y6B0,<K0
M]:7]1?Y1HY&C8$BE_5&CD:.I_X7]A?ZI@$6CA:/0 N:D:.5HL -E9\C*T,I,
M38QE9TBE_47^4:.1HZ7^A?UHR,K0LV"F9Z5G2D;^\!#E:) QRM#UI?U%_A&C
MD:-@2*7]$:.1HZG_A?V%_JF 1:.%H] "YJ1HY6BP V5GR,K0RDR@C&5G2*7]
M1?X1HY&CI?Z%_6C(RM"S8*50\%.F9Z5G2D;^\!#E:) QRM#UI?U%_E&CD:-@
M2*7]4:.1HZG_A?V%_JF 1:.%H] "YJ1HY6BP V5GB,K0RDSWC&5G2*7]1?Y1
MHY&CI?Z%_6B(RM"S8*9GI6=*1O[P$.5HD#'*T/6E_47^$:.1HV!(I?T1HY&C
MJ?^%_87^J8!%HX6CT +FI&CE:+ #96>(RM#*3$J-96=(I?U%_A&CD:.E_H7]
M:(C*T+-@I/R] (^%_4I%_87]Q/ZP?J50\#JF:/ ,I6A*.,CE9Y $RM#X8&5H
M2*7]4:.1HVA&_3CP!LK0Y4RQC4BI@(7]1:.%H] "YJ1HRM#13+&-IFCP%*5H
M2CA(I?T1HY&C:,CE9Y *RM#PI?T1HY&C8&5H1OTX\ ;*T-],\XU(J8"%_46C
MA:/0 N:D:,K0RTSSC:50\#JF:/ ,I6A*.(CE9Y $RM#X8&5H2*7]4:.1HVA&
M_3CP!LK0Y4POCDBI@(7]1:.%H] "YJ1HRM#13"^.IFCP%*5H2CA(I?T1HY&C
M:(CE9Y *RM#PI?T1HY&C8&5H1OTX\ ;*T-],<8Y(J8"%_46CA:/0 N:D:,K0
MRTQQCJT8T"GUC1C08%-024Y!3"!#4D%#2T52(%-,2B V+SDU
M
M                    #_?S\?#P<# ?]_/Q\/!P,!_W\_'P\' P'_?S\?
M#P<# ?]_/Q\/!P,!_W\_'P\' P'_?S\?#P<# ?]_/Q\/!P,!_W\_'P\' P'_
M?S\?#P<# ?]_/Q\/!P,!_W\_'P\' P'_?S\?#P<# ?]_/Q\/!P,!_W\_'P\'
M P'_?S\?#P<# 1H:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:
M&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:
M&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:
!&AH:

end

begin 666 shape3.2
M 1PA'   CR!#54)%,T0@4TA!4$4@24Y)5"!04D]'4D%- $<<!0"7-3$L,C4U
M.I<U,BPQ,C<ZES4U+#(U-3J7-38L,3(W.IP ;AQB (\@34%)3B!04D]'4D%-
M(%-405)44R!!5"!,24Y%(#4P,#  =!QC #H B!QD (\@4%)/1U)!32!.3U1%
M4P":'&X CR!33$H@-B\Q-B\Y-0"P''@ CR!$051!($9/4DU!5"!)4SH UAQ]
M (\@3E5-4$])3E13+%@Q+%DQ+%HQ+%@R+%DR+%HR+"XN+@#W'(( CR!42$4@
M4$],64=/3B!,25-4("I-55-4*B!"10 5'8< CR!415)-24Y!5$5$(%=)5$@@
M02!:15)/(0 ;'8D .@ ^'8P CR!/1B!#3U524T4L(%1(15)%($E3($$@3$]4
M($]& %X=D0"/($154$Q)0T%424].($1205=)3D<@3$E.15, @QV3 (\@5TA%
M3B!&04-%4R!!4D5.)U0@0D5)3D<@1DE,3$5$ *8=E@"/($$@1D%35"!04D]'
M4D%-(%=/54Q$3B=4($1205< RAV@ (\@5$A%(%-!344@3$E.12!45TE#12P@
M0E54(%1(25, Z!VE (\@25,@1T]/1"!%3D]51T@@1D]2($Y/5R$ "AZG (\@
M3T8@0T]54E-%+"!&3U(@1DE,3$5$($9!0T53 "L>J "/(%1(25,@25,@3D\@
M3$].1T52($%.($E34U5% #$>J@ Z %8>M "/($9%14P@1E)%12!43R!0550@
M64]54B!/5TX@1$%400!\'KD CR!)3BP@0E54($-(04Y'12!42$4@4$]+12!)
M3B!,24Y% )@>O@"/(#4P,#4@5$\@55-%($,Q($%.1"!#,@">'L@ .@##'M(
MCR!33U)262!!0D]55"!33TU%($]&(%1(12!,25143$4 WA[7 (\@0E5'4RP@
M15(N+BX@1D5!5%5215, _A[< (\@5$A)4R!705,@02!214%,(%)54T@@2D]"
M+@ $'^8 .@ G'_  CR!!3D0@248@64]5($9%14P@4T\@24Y#3$E.140L $0?
M^@"/(%=2251%(%1/(%-*541$0$Y752Y%1%4 2A\$ 3H :1\. 8\@04Y$($%"
M3U9%($%,3"P@2$%612!&54XA &\?& $Z ($?Y@./($9)4E-4(%-(05!% )<?
MYP-!,;+"*#8Q*3I!,K+"*#8R*0#''^@#@R T+"TR-BPM,C8L-C0L,C8L+3(V
M+#8T+#(V+#8T+#8T+"TR-BPV-"PV- #S'_(#@R T+"TR-BPM,C8L,"PR-BPM
M,C8L,"PR-BPV-"PP+"TR-BPV-"PP !<@_ .#(#,L,"PM,C8L,S(L,38L,S(L
M,S(L+3$V+#,R+#,R !\@!@2#(#  ,B!*!(\@4T5#3TY$(%-(05!% $@@2P1"
M,;+"*#8Q*3I",K+"*#8R*0!\($P$@R T+"TQ-2PS,"PM,3(L+3$U+"TS,"PM
M,3(L+34W+"TS,"PM,RPM-3$L+38L+3, K"!6!(,@-"PQ-2PS,"PM,3(L-3$L
M+38L+3,L-3<L+3,P+"TS+#$U+"TS,"PM,3( X"!@!(,@-"PQ-2PS,"PM,3(L
M,34L+3,P+"TQ,BPM,34L+3,P+"TQ,BPM,34L,S L+3$R "0A:@2#(#<L,34L
M+3,P+"TQ,BPU-RPM,S L+3,L,S,L+3,P+#8L,"PM,S L,3(L+3,S+"TS,"PV
M+"TU-RPM,S L+3, -B%L!(,@+3$U+"TS,"PM,3( 6R%T!(,@,RPS,RPM,S L
M,"PS,RPM,S L+30L,SDL+3,P+"TR (,A?@2#(#,L+3,S+"TS,"PP+"TS.2PM
```

```
M,S L+3(L+3,S+"TS,"PM- "O(8@$@R T+#8L+3,P+#,L-BPM,S L+38L,3@L
M+3,P+"TT+#$X+"TS,"PQ -\AD@2#(#0L+38L+3,P+#,L+3$X+"TS,"PQ+"TQ
M."PM,S L+30L+38L+3,P+"TV  <BG 2#(#,L+34W+"TS,"PM,RPM,S,L+3,P
M+#8L+34Q+"TQ,BPM,P O(J8$@R S+"TS,RPM,S L-BPM,34L,S L+3$R+"TU
M,2PM,3(L+3, 4R*P!(,@,RPM,34L,S L+3$R+"TS,RPM,S L-BPP+#8L,3(
M=2*Z!(,@,RPP+#8L,3(L+3,S+"TS,"PV+# L+3,P+#$R )DBQ 2#(#,L,34L
M,S L+3$R+"TQ-2PS,"PM,3(L,"PV+#$R +HBS@2#(#,L,"PM,S L,3(L,S,L
M+3,P+#8L,"PV+#$R -PBV 2#(#,L,"PV+#$R+#,S+"TS,"PV+#$U+#,P+"TQ
M,@ !(^($@R S+#$U+#,P+"TQ,BPS,RPM,S L-BPU,2PM,3(L+3, )B/L!(,@
M,RPS,RPM,S L-BPU-RPM,S L+3,L-3$L+3$R+"TS "XC]@2#(#  0",2!8\@
M5$A)4D0@4TA!4$4 5B,3!4,QLL(H-C$I.D,RLL(H-C(I ' C% 6/(%E/55(@
M1$%402!'3T53($A%4D4 >"/0!X,@, ",(X@30E"R,C4VK"@YK#$VJC8I + C
MC!./($-(04Y'12!"14Q/5R!43R!03TE.5"!43R!$051! ,(CC1.7-C4L0C$Z
MES8V+$(R .$CDA.'3CJ70E L3CI"4+)"4*HQ.HM.LC"G-3 U, #I(Y<3F2!.
M  8DG!.!2;(QI#.L3CJ'4#J+4+,PIU"R,C4VJE  &"2F$Y="4"Q0.D)0LD)0
MJC$ (B2K$YDB+B([ "XDL!.".HDU,#$P #\DNA.>,S(W-C@ZG#J,.H     :
M&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:
0&AH:&AH:&AH:&AH:&AH:&AH:

end

begin 666 tables$8f80-$95ff
M@(\  @,%!P@*"PT/$!$3%!47&!D:&QP='1X>'Q\@(" @(" @'Q\>'AT='!L:
M&1@7%103$1 /#0L*" <% P( _OW[^?CV]?/Q\._M[.OIZ.?FY>3CX^+BX>'@
MX.#@X.#@X>'BXN/CY.7FY^CIZ^SM[_#Q\_7V^/G[_?X _P    #_\" @(" ?
M'QX>'1T<&QH9&!<5%!,1$ \-"PH(!P4# @#^_?OY^/;U\_'P[^WLZ^GHY^;E
MY./CXN+AX>#@X.#@X.#AX>+BX^/DY>;GZ.GK[.WO\/'S]?;X^?O]_@ " P4'
M" H+#0\0$1,4%1<8&1H;'!T='AX?'R @("  _____P  (B(B(B(C(R,C(R,C
M(R,D)"0D)"0D)"4E)24E)24E)B8F)B8F)B8G)R<G)R<G*"@H*"@H*"DI*2DI
M*2HJ*BHJ*BLK*RLK*RPL+"PL+"TM+2TM+BXN+BXN+R\O+S P,# P,3$Q,3$R
M,C(R,S,S,S0T-#0U-34U-C8V-C<W-S@X.#@8&!@8&1D9&1D9&1D9&1D9&1D9
M&1D9&AH:&AH:&AH:&AH:&AH:&AL;&QL;&QL;&QL;&QP<'!P<'!P<'!P<
M'!P<'1T='1T='1T='1T='1X>'AX>'AX>'AX>'A\?'Q\?'Q\?'Q\?(" @(" @
M(" @(" A(2$A(2$A(2$A(B(B(O__     /____\ \   _P#__P    #_____
M     /____\     _____P    #_ /__     /_P__\     _____P    #_
M____     /____\     _____P    #_____     /____\     _____P
M  #_____     /_P                0$! 0$! 0$" @(" @(# P,#! 0$
M! 4%!04&!@8'!P<(" @)"0D*"@L+"PP,#0T.#@\/$! 1$1(2$Q,4%!45%A<7
M&!@9&AH;'!P='AX?(" A(B,C)"4F)B<H*2DJ*RPM+BXO,#$R,S0U-38W.#DZ
M.SP]/C] 04)#1$5&1TA)2DM-3D]045)35%976"LJ*2DH)R8F)20C(R(A(" ?
M'AX='!P;&AH9&!@7%Q85%104$Q,2$A$1$! /#PX.#0T,# L+"PH*"0D)" @(
M!P<'!@8&!04%!00$! 0# P,# @(" @(" 0$! 0$! 0$
M            $! 0$! 0$! @(" @(" P,# P0$! 0%!04%!@8&!P<'" @(
M"0D)"@H+"PL,# T-#@X/#Q 0$1$2$A,3%!05%187%Q@8&1H:&QP<'1X>'R @
M(2(C(R0E)B8G*"DI*BLL+2XN+S Q,C,T-34V-S@Y.CL\/3X_0$%"0T1%1D=(
M24I+34Y/4%%24U165U@K*BDI*"<F)B4D(R,B(2 @'QX>'1P<&QH:&1@8%Q<6
M%144%!,3$A(1$1 0#P\.#@T-# P+"PL*"@D)"0@(" <'!P8&!@4%!04$! 0$
M P,# P(" @(" @$! 0$! 0$!                                  ! 0$!
M 0$! 0(" @(" @,# P,$! 0$!04%!08&!@<'!P@(" D)"0H*"PL+# P-#0X.
M#P\0$!$1$A(3$Q04%146%Q<8&!D:&AL<'!T>'A\@("$B(R,D)28F)R@I*2HK
M+"TN+B\P,3(S-#4U-C<X.3H[/#T^/T _/CT\.SHY.#<V-34T,S(Q,"\N+BTL
M*RHI*2@G)B8E)",C(B$@(!\>'AT<'!L:&AD8&!<7%A45%!03$Q(2$1$0$ \/
M#@X-#0P,"PL+"@H)"0D(" @'!P<&!@8%!04%! 0$! ,# P," @(" @(! 0$!
M 0$! 0                                  0$! 0$! 0$" @(" @(# P,#
M! 0$! 4%!04&!@8'!P<(" @)"0D*"@L+"PP,#0T.#@\/$! 1$1(2$Q,4%!45
M%A<7&!@9&AH;'!P='AX?(" A(B,C)"4F)B<H*2DJ*RPM+BXO,#$R,S0U-38W
M.#DZ.SP]/C] /SX]/#LZ.3@W-C4U-#,R,3 O+BXM+"LJ*2DH)R8F)20C(R(A
M(" ?'AX='!P;&AH9&!@7%Q85%104$Q,2$A$1$! /#PX.#0T,# L+"PH*"0D)
M" @(!P<'!@8&!04%!00$! 0# P,# @(" @(" 0$! 0$! 0$
M !H:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:
M&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:
E&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:

end


**********************************
*``````````````````````````````````*
*`Stephen`Judd`````````````````````*
*`George`Taylor```````````````````*
*`Started:`7/11/94```````````````*
*`Finished:`7/19/94```````````````*
*`v2.0`Completed:`12/17/94`````*
*`v3.0`Completed:`3/20/95``````*
*`v3.1`Completed:`6/14/95``````*
*`v3.2`Completed:`6/15/95``````*
*``````````````````````````````````*
*`Well,`if`all`goes`well`this``*
*`program`will`rotate`a`cube.``*
*``````````````````````````````````*
*`v2.0`+`New`and`Improved!`````*
```

```
*`Now`with`faster`routines,````*
*`hidden`surfaces,`filled`````*
*`faces,`and`extra`top`secret``*
*`text`messages!`````````````*
*`````````````````````````````*
*`v3.0`+`Fast`chunky`line`````*
*`routine.`````````````````````*
*`````````````````````````````*
*`v3.1`+`General`polygon`plot``*
*`with`hidden`faces`(X-product)*
*`and`zoom`feature.`````````````*
*`````````````````````````````*
*`v3.2`+`EOR-buffer`filling````*
*`````````````````````````````*
*`This`program`is`intended`to``*
*`accompany`the`article`in````*
*`C=Hacking,`Jun.`95`issue.````*
*`For`details`on`this`program,`*
*`read`the`article!`````````````*
*`````````````````````````````*
*`Write`to`us!``````````````````*
*`````````````````````````````*
*`Myself`when`young`did`````````*
*`eagerly`frequent`````````````*
*`Doctor`and`Saint,`and`heard``*
*`great`Argument`````````````````*
*``About`it`and`about:`but`````*
*``evermore`````````````````````*
*`Came`out`by`the`same`Door````*
*`as`in`I`went.`````````````````*
*````-``Rubaiyat`````````````````*
*`````````````````````````````*
*`Though`I`speak`with`the`````*
*`tongues`of`men`and`of`angles`*
*`and`have`not`love,`I`am`````*
*`become`as`sounding`brass,`or`*
*`a`tinkling`cymbal.`````````*
*````-`1`Corinthians`13````````*
*`````````````````````````````*
*`P.S.`This`was`written`using``*
*``````Merlin`128.`````````````*
*******************************

 ORG $8000

*`Constants

BUFF1 EQU $3000 ;First`character`set
BUFF2 EQU $3800 ;Second`character`set
EORBUF EQU $4000 ;EOR-buffer
BUFFER EQU $A3 ;Presumably`the`tape`won't`be`running
X1 EQU $FB ;Points`for`drawing`a`line
Y1 EQU $FC ;These`zero`page`addresses
X2 EQU $FD ;don't`conflict`with`BASIC
Y2 EQU $FE
OLDX EQU $FD
CHUNK EQU $FE
DX EQU $67 ;This`is`shared`with`T1`below
DY EQU $68
TEMP1 EQU $FB ;Of`course,`could`conflict`with`x1
TEMP2 EQU $FC ;Temporary`variables
ZTEMP EQU $02 ;Used`for`buffer`swap.``Don't`touch.
Z1 EQU $22 ;Used`by`math`routine
Z2 EQU $24 ;Don't`touch`these`either!
Z3 EQU $26
Z4 EQU $28
K EQU $B6 ;Constant`used`for`hidden
   ;surface`detection`-`don't`touch
HIDE EQU $B5 ;Are`surfaces`hidden?
FILL EQU $50 ;Are`we`using`EOR-fill?
ANGMAX EQU 120 ;There`are`2*pi/angmax`angles

*`VIC

VMCSB EQU $D018
BKGND EQU $D020
BORDER EQU $D021
SSTART EQU 1344 ;row`9`in`screen`memory`at`1024


*`Kernal
```

```
CHROUT EQU $FFD2
GETIN EQU $FFE4

*`Some`variables

GLOBXMIN = $3F ;These`are`used`in`clearing`the
GLOBXMAX = $40 ;drawing`(global)`buffer
GLOBYMIN = $41
GLOBYMAX = $42
LOCXMIN = $57 ;These`are`used`in`clearing`the
LOCXMAX = $58 ;EOR`(local)`buffer
LOCYMIN = $59
LOCYMAX = $60
P1X = $92 ;These`are`temporary`storage
P1Y = $93 ;Used`in`plotting`the`projection
P1Z = $94
P2X = $95 ;They`are`here`so`that`we
P2Y = $96 ;don't`have`to`recalculate`them.
P2Z = $AE
P3X = $AF ;They`make`life`easy.
P3Y = $B0
P3Z = $B1 ;Why`are`you`looking`at`me`like`that?
P1T = $B2 ;Don't`you`trust`me?
P2T = $B3
P3T = $B4 ;Having`another`child`wasn't`my`idea.
INDEX = $51
COUNTPTS = $52
ZOOM = $71 ;Zoom`factor
DSX = $61 ;DSX`is`the`increment`for
   ;rotating`around`x
DSY = $62 ;Similar`for`DSY,`DSZ
DSZ = $63
SX = $64 ;These`are`the`actual`angles`in`x`y`and`z
SY = $65
SZ = $66
T1 = $67 ;These`are`used`in`the`rotation
T2 = $68
T3 = $69 ;See`the`article`for`more`details
T4 = $6A
T5 = $6B
T6 = $6C
T7 = $6D
T8 = $6E
T9 = $6F
T10 = $70
A11 = $A5 ;These`are`the`elements`of`the`rotation`matrix
B12 = $A6 ;XYZ
C13 = $A7
D21 = $A8 ;The`number`denotes`(row,column)
E22 = $A9
F23 = $AA
G31 = $AB
H32 = $AC
I33 = $AD


***`Macros

MOVE MAC
 LDA ]1
 STA ]2
 <<<

GETKEY MAC  ;Wait`for`a`keypress
WAIT JSR GETIN
 CMP #00
 BEQ WAIT
 <<<

DEBUG MAC  ;Print`a`character
 DO`0``;Don't`assemble

 LDA`#]1
 JSR`CHROUT
 CLI
 >>> GETKEY ;And`wait`to`continue
 CMP #'s' ;My`secrect`switch`key
 BNE L1
 JSR CLEANUP
 JMP DONE
```

```
L1 CMP #'x' ;My`secret`abort`key
 BNE DONE
 JMP CLEANUP
 FIN
DONE <<<

DEBUGA MAC
 DO`0
 LDA ]1
 STA 1024
 FIN
DONEA <<<

*-------------------------------

 LDA #$00
 STA BKGND
 STA BORDER
 LDA VMCSB
 AND #%00001111 ;Screen`memory`to`1024
 ORA #%00010000
 STA VMCSB

 LDY #00
 LDA #<TTEXT
 STA TEMP1
 LDA #>TTEXT
 STA TEMP2
 JMP TITLE
TTEXT HEX 9305111111 ;clear`screen,`white,`crsr`dn
 TXT '````````````cube3d`v3.2',0d,0d
 TXT '````````````````by',0d
 HEX 9F ;cyan
 TXT '````stephen`judd'
 HEX 99
 TXT '````george`taylor',0d,0d
 HEX 9B
 TXT '``check`out`the`jan.`95`issue`of',0d
 HEX 96
 TXT '``c=hacking'
 HEX 9B
 TXT ''for`more`details!',0d
 HEX 0D1D1D9E12
 TXT 'f1/f2',92
 TXT ''-`inc/dec`x-rotation',0d
 HEX 1D1D12
 TXT 'f3/f4',92
 TXT ''-`inc/dec`y-rotation',0d
 HEX 1D1D12
 TXT 'f5/f6',92
 TXT ''-`inc/dec`z-rotation',0d
 HEX 1D1D12
 TXT ''f7``',92
 TXT ''-`reset',0d
 HEX 1D1D12
 TXT ''+/-`',92
 TXT ''-`zoom`in/out',0d
 HEX 1D1D12
 TXT '``h``',92
 TXT ''-`toggle`hidden`surfaces',0d
 HEX 1D1D12
 TXT 'space',92
 TXT ''-`toggle`surface`filling',0d,0d
 TXT '``press`q`to`quit',0d
 HEX 0D05
 TXT ''''''press`any`key`to`begin',0d
 HEX 00
TITLE LDA (TEMP1),Y
 BEQ :CONT
 JSR CHROUT
 INY
 BNE TITLE
 INC TEMP2
 JMP TITLE
:CONT >>> GETKEY

****`Set`up`tables(?)

*`Tables`are`currently`set`up`in`BASIC
*`and`by`the`assembler.
```

```
TABLES LDA #>TMATH1
 STA Z1+1
 STA Z2+1
 LDA #>TMATH2
 STA Z3+1
 STA Z4+1

****`Clear`screen`and`set`up`"bitmap"
SETUP LDA #$01 ;White
 STA $D021 ;This`is`done`so`that`older
 LDA #147 ;machines`will`set`up
 JSR CHROUT
 LDA #$00 ;correctly
 STA $D021
 LDA #<SSTART
 ADC #12 ;The`goal`is`to`center`the`graphics
 STA TEMP1 ;Column`12
 LDA #>SSTART ;Row`9
 STA TEMP1+1 ;SSTART`points`to`row`9
 LDA #00
 LDY #00
 LDX #00 ;x`will`count`16`rows`for`us
 CLC

:LOOP STA (TEMP1),Y
 INY
 ADC #16
 BCC :LOOP
 CLC
 LDA TEMP1
 ADC #40 ;Need`to`add`40`to`the`base`pointer
 STA TEMP1 ;To`jump`to`the`next`row
 LDA TEMP1+1
 ADC #00 ;Take`care`of`carries
 STA TEMP1+1
 LDY #00
 INX
 TXA  ;X`is`also`an`index`into`the`character`number
 CPX #16
 BNE :LOOP ;Need`to`do`it`16`times

****`Clear`buffers

 LDA #<BUFF1
 STA BUFFER
 LDA #>BUFF1
 STA BUFFER+1
 LDY #$00
 LDX #24 ;Assuming`all`three`buffers`are
 LDA #$00 ;back-to-back
:BLOOP STA (BUFFER),Y
 INY
 BNE :BLOOP
 INC BUFFER+1
 DEX
 BNE :BLOOP

****`Set`up`buffers

 LDA #<BUFF1
 STA BUFFER
 LDA #>BUFF1
 STA BUFFER+1
 STA ZTEMP ;ztemp`will`make`life`simple`for`us
 LDA VMCSB
 AND #%11110001 ;Start`here`so`that`swap`buffers`will`work`right
 ORA #%00001110
 STA VMCSB

****`Set`up`initial`values

INIT LDA #00
 STA LOCXMIN
 STA LOCXMAX
 STA LOCYMIN
 STA LOCYMAX
 STA GLOBXMIN
 STA GLOBYMIN
 STA GLOBXMAX
 STA GLOBYMAX
 STA DSX
```

```
 STA DSY
 STA DSZ
 STA SX
 STA SY
 STA SZ
 STA FILL
 LDA #01
 STA HIDE
 LDA #64
 STA ZOOM

*--------------------------------
*`Main`loop

****`Get`keypress

MAIN
 CLI
KPRESS JSR GETIN
 CMP #133 ;F1?
 BNE :F2
 LDA DSX
 CMP #ANGMAX/2 ;No`more`than`pi
 BEQ :CONT1
 INC DSX ;otherwise`increase`x-rotation
 JMP :CONT
:F2 CMP #137 ;F2?
 BNE :F3
 LDA DSX
 BEQ :CONT1
 DEC DSX
 JMP :CONT
:F3 CMP #134
 BNE :F4
 LDA DSY
 CMP #ANGMAX/2
 BEQ :CONT1
 INC DSY ;Increase`y-rotation
 JMP :CONT
:F4 CMP #138
 BNE :F5
 LDA DSY
 BEQ :CONT1
 DEC DSY
 JMP :CONT
:F5 CMP #135
 BNE :F6
 LDA DSZ
 CMP #ANGMAX/2
 BEQ :CONT1
 INC DSZ ;z-rotation
 JMP :CONT
:F6 CMP #139
 BNE :F7
 LDA DSZ
 BEQ :CONT1
 DEC DSZ
 JMP :CONT
:F7 CMP #136
 BNE :PLUS
 JMP INIT
:CONT1 JMP :CONT
:PLUS CMP #'+'
 BNE :MINUS
 INC ZOOM ;Bah,`who`needs`error`checking?
 INC ZOOM
 JMP :CONT
:MINUS CMP #'-'
 BNE :H
 DEC ZOOM
 DEC ZOOM
 BPL :CONT
 INC ZOOM
 INC ZOOM
 JMP :CONT
:H CMP #'h'
 BNE :SPACE
 LDA HIDE
 EOR #$01
 STA HIDE
 JMP :CONT
```

```
:SPACE CMP #'`'
 BNE :Q
 LDA FILL
 EOR #$01
 STA FILL
 JMP :CONT
:Q CMP #'q' ;q`quits
 BNE :CONT
 JMP CLEANUP

:CONT SEI  ;Speed`things`up`a`bit

****`Update`angles

UPDATE CLC
 LDA SX
 ADC DSX
 CMP #ANGMAX ;Are`we`>=`maximum`angle?
 BCC :CONT1
 SBC #ANGMAX :If so, reset
:CONT1 STA SX
 CLC
 LDA SY
 ADC DSY
 CMP #ANGMAX
 BCC :CONT2
 SBC #ANGMAX ;Same`deal
:CONT2 STA SY
 CLC
 LDA SZ
 ADC DSZ
 CMP #ANGMAX
 BCC :CONT3
 SBC #ANGMAX
:CONT3 STA SZ

****`Rotate`coordinates

ROTATE

***`First,`calculate`t1,t2,...,t10

**`Two`macros`to`simplify`our`life
ADDA MAC  ;Add`two`angles`together
 CLC
 LDA ]1
 ADC ]2
 CMP #ANGMAX ;Is`the`sum`>`2*pi?
 BCC DONE
 SBC #ANGMAX ;If`so,`subtract`2*pi
DONE <<<

SUBA MAC  ;Subtract`two`angles
 SEC
 LDA ]1
 SBC ]2
 BCS DONE
 ADC #ANGMAX ;Oops,`we`need`to`add`2*pi
DONE <<<

**`Now`calculate`t1,t2,etc.

 >>> SUBA,SY;SZ
 STA T1 ;t1=sy-sz
 >>> ADDA,SY;SZ
 STA T2 ;t2=sy+sz
 >>> ADDA,SX;SZ
 STA T3 ;t3=sx+sz
 >>> SUBA,SX;SZ
 STA T4 ;t4=sx-sz
 >>> ADDA,SX;T2
 STA T5 ;t5=sx+t2
 >>> SUBA,SX;T1
 STA T6 ;t6=sx-t1
 >>> ADDA,SX;T1
 STA T7 ;t7=sx+t1
 >>> SUBA,T2;SX
 STA T8 ;t8=t2-sx
 >>> SUBA,SY;SX
 STA T9 ;t9=sy-sx
 >>> ADDA,SX;SY
```

```
 STA T10 ;t10=sx+sy

*`Et`voila!

***`Next,`calculate`A,B,C,...,I

**`Another`useful`little`macro
DIV2 MAC  ;Divide`a`signed`number`by`2
;It`is`assumed`that`the`number
 BPL POS ;is`in`the`accumulator
 CLC
 EOR #$FF ;We`need`to`un-negative`the`number
 ADC #01 ;by`taking`it's`complement
 LSR  ;divide`by`two
 CLC
 EOR #$FF
 ADC #01 ;Make`it`negative`again
 JMP DONEDIV
POS LSR  ;Number`is`positive
DONEDIV <<<

MUL2 MAC  ;Multiply`a`signed`number`by`2
 BPL POSM
 CLC
 EOR #$FF
 ADC #$01
 ASL
 CLC
 EOR #$FF
 ADC #$01
 JMP DONEMUL
POSM ASL
DONEMUL <<<

**`Note`that`we`are`currently`making`a`minor`leap
**`of`faith`that`no`overflows`will`occur.

:CALCA CLC
 LDX T1
 LDA COS,X
 LDX T2
 ADC COS,X
 STA A11 ;A=(cos(t1)+cos(t2))/2
:CALCB LDX T1
 LDA SIN,X
 SEC
 LDX T2
 SBC SIN,X
 STA B12 ;B=(sin(t1)-sin(t2))/2
:CALCC LDX SY
 LDA SIN,X
 >>> MUL2
 STA C13 ;C=sin(sy)
:CALCD SEC
 LDX T8
 LDA COS,X
 LDX T7
 SBC COS,X
 SEC
 LDX T5
 SBC COS,X
 CLC
 LDX T6
 ADC COS,X ;Di=(cos(t8)-cos(t7)+cos(t6)-cos(t5))/2
 >>> DIV2
 CLC
 LDX T3
 ADC SIN,X
 SEC
 LDX T4
 SBC SIN,X
 STA D21 ;D=(sin(t3)-sin(t4)+Di)/2
:CALCE SEC
 LDX T5
 LDA SIN,X
 LDX T6
 SBC SIN,X
 SEC
 LDX T7
 SBC SIN,X
 SEC
```

```
 LDX T8
 SBC SIN,X ;Ei=(sin(t5)-sin(t6)-sin(t7)-sin(t8))/2
 >>> DIV2
 CLC
 LDX T3
 ADC COS,X
 CLC
 LDX T4
 ADC COS,X
 STA E22 ;E=(cos(t3)+cos(t4)+Ei)/2
:CALCF LDX T9
 LDA SIN,X
 SEC
 LDX T10
 SBC SIN,X
 STA F23 ;F=(sin(t9)-sin(t10))/2
:CALCG LDX T6
 LDA SIN,X
 SEC
 LDX T8
 SBC SIN,X
 SEC
 LDX T7
 SBC SIN,X
 SEC
 LDX T5
 SBC SIN,X ;Gi=(sin(t6)-sin(t8)-sin(t7)-sin(t5))/2
 >>> DIV2

 CLC
 LDX T4
 ADC COS,X
 SEC
 LDX T3
 SBC COS,X
 STA G31 ;G=(cos(t4)-cos(t3)+Gi)/2
:CALCH CLC
 LDX T6
 LDA COS,X
 LDX T7
 ADC COS,X
 SEC
 LDX T5
 SBC COS,X
 SEC
 LDX T8
 SBC COS,X ;Hi=(cos(t6)+cos(t7)-cos(t5)-cos(t8))/2
 >>> DIV2
 CLC
 LDX T3
 ADC SIN,X
 CLC
 LDX T4
 ADC SIN,X
 STA H32 ;H=(sin(t3)+sin(t4)+Hi)/2
:WHEW CLC
 LDX T9
 LDA COS,X
 LDX T10
 ADC COS,X
 STA I33 ;I=(cos(t9)+cos(t10))/2

**`It's`all`downhill`from`here.

DOWNHILL
****`Clear`buffer
*`A`little`macro

SETBUF MAC  ;Put`buffers`where`they`can`be`hurt
 LDA #00
 STA BUFFER
 LDA ZTEMP ;High`byte
STABUF STA BUFFER+1
 <<<

 >>> SETBUF
CLRDRAW LDX #08
 LDA #00
:FOOL LDY #00
:DOPE STA (BUFFER),Y
 INY
```

```
 BNE :DOPE
 INC BUFFER+1
 DEX
 BNE :FOOL

****`My`goodness`but`I'm`a`dope
*CLRDRAW`LDA`GLOBXMIN
*`LSR``;Need`to`get`into`the`right`column
*`BCC`:EVEN`;Explained`in`more`detail`below
*`LDY`#$80
*`STY`BUFFER`;Presumably`this`will`be`a`little
*`CLC``;more`efficient.
*:EVEN`ADC`BUFFER+1
*`STA`BUFFER+1
*`LDA`GLOBXMAX
*`SEC
*`SBC`GLOBXMIN
*`TAX
*`INX
*`LDY`GLOBYMAX
*`BEQ`:RESET
*:YAY`LDA`#$00
*`LDY`GLOBYMAX
*:BLAH`STA`(BUFFER),Y
*`DEY
*`CPY`GLOBYMIN
*`BCS`:BLAH
*`LDA`BUFFER
*`EOR`#$80
*`STA`BUFFER
*`BNE`:WHOPEE
*`INC`BUFFER+1
*:WHOPEE`DEX
*`BNE`:YAY
*:RESET`LDA`#0`;Need`to`reset`these`guys
*`STA`GLOBXMAX
*`STA`GLOBYMAX
*`LDA`#$FF
*`STA`GLOBXMIN
*`STA`GLOBYMIN

****`Next,`read`and`draw`polygons

READDRAW LDY #00
 STY INDEX
OBJLOOP LDY INDEX
 LDA POLYLIST,Y ;First,`the`number`of`points
 BNE :CONT ;But`if`numpoints`is`zero`then
 JMP OBJDONE ;we`are`at`the`end`of`the`list
:CONT STA COUNTPTS
 INC INDEX

*`Rotate`project`and`draw`the`polygon
*`Make`sure`buffer`being`drawn`to`is`clear!

:DOIT JSR ROTPROJ

*`Convert`xmin`and`xmax`to`columns

 LDA LOCXMIN
 LSR
 LSR
 LSR  ;x`mod`8
 STA LOCXMIN
 CMP GLOBXMIN
 BCS :NAH
 STA GLOBXMIN
:NAH LDA LOCYMIN
 CMP GLOBYMIN
 BCS :UHUH
 STA GLOBYMIN
:UHUH LDA LOCXMAX
 LSR
 LSR
 LSR
 STA LOCXMAX
 CMP GLOBXMAX
 BCC :NOWAY
 STA GLOBXMAX
:NOWAY LDA LOCYMAX
 CMP GLOBYMAX
```

```
 BCC EORFILL
 STA GLOBYMAX

*`If`using`the`EOR-buffer,`copy`into`drawing`buffer
*`And`then`clear`the`EOR-buffer

EORFILL LDA FILL
 BEQ OBJLOOP

 >>> SETBUF
 LDA #<EORBUF
 STA TEMP1
 LDA #>EORBUF
 STA TEMP1+1

 LDA LOCXMIN ;LOCXMIN`now`contains`column
 LSR  ;Each`column`is`128`bytes
 BCC :EVEN ;So`there`might`be`a`carry
 LDY #$80
 STY BUFFER
 STY TEMP1
 CLC
:EVEN STA T2
 ADC BUFFER+1
 STA BUFFER+1 ;Each`column`is`128`bytes
 LDA T2
 ADC TEMP1+1 ;Now`we`will`start`at`the
 STA TEMP1+1 ;column

 LDA LOCXMAX
 SEC
 SBC LOCXMIN
 TAX ;Total`number`of`columns`to`do
 INX  ;e.g.`fill`columns`1..3
 LDY LOCYMAX
 BNE :FOOP
 INC LOCYMAX
:FOOP LDY LOCYMAX
 LDA #00
:GOOP EOR (TEMP1),Y ;EOR-buffer
 PHA
*`Maybe`put`an`EOR`below?
 EOR (BUFFER),Y
 STA (BUFFER),Y
 LDA #00 ;Might`as`well`clear`it`now
 STA (TEMP1),Y
 PLA
 DEY
 CPY LOCYMIN
 BCS :GOOP
 LDA BUFFER
 EOR #$80
 STA BUFFER
 STA TEMP1
 BNE :BOOP
 INC BUFFER+1
 INC TEMP1+1
:BOOP DEX
 BNE :FOOP
 JMP OBJLOOP

OBJDONE
****`Swap`buffers

SWAPBUF LDA VMCSB
 EOR #$02 ;Pretty`tricky,`eh?
 STA VMCSB
 LDA #$08
 EOR ZTEMP ;ztemp=high`byte`just`flips
 STA ZTEMP ;between`$30`and`$38

 JMP MAIN ;Around`and`around`we`go...

 TXT 'Gee`Brain,`what`do`you`want`to`do`'
 TXT 'tonight?'

**`Rotate,`project,`and`store`the`points
*
*`This`part`is`a`significant`change`since
*`v2.0.``Now`it`is`a`completely`general`polygon`plotter.
*`A`set`of`points`is`read`in,`rotated`and`projected,`and
```

```
*`plotted`into`the`drawing`buffer`(EOR`or`normal).

ROTPROJ

*`A`neat`macro
NEG MAC  ;Change`the`sign`of`a`two's`complement
 CLC
 LDA ]1 ;number.
 EOR #$FF
 ADC #$01
 <<<

*--------------------------------
*`These`macros`replace`the`previous`projection
*`subroutine.

SMULT`MAC`;Multiply`two`signed`8-bit
 ;numbers:`A*Y/64`->`A
 STA`Z3
 CLC``;This`multiply`is`for`normal
 EOR`#$FF`;numbers,`i.e.`x=-64..64
 ADC`#$01
 STA`Z4
 LDA`(Z3),Y
 SEC
 SBC`(Z4),Y
 <<<``;All`done`:)

SMULTZ MAC ;Multiply`two`signed`8-bit
   ;numbers:`A*Y/64`->`A
 STA Z1
 CLC  ;And`this`multiply`is`specifically
 EOR #$FF ;for`the`projection`part,`where
 ADC #$01 ;numbers`are`-110..110`and`0..40
 STA Z2
 LDA (Z1),Y
 SEC
 SBC (Z2),Y
 <<<  ;All`done`:)

PROJECT MAC  ;The`actual`projection`routine
;The`routine`takes`the`point
;]1`]2`]3,`rotates`and
;projects`it,`and`stores`the
;result`in`]1`]2`]3.

 LDY ]1 ;Multiply`first`rotation`column
 LDA A11
 >>> SMULT
 STA P1T
 LDA D21
 >>> SMULT
 STA P2T
 LDA G31
 >>> SMULT
 STA P3T
 LDY ]2 ;Second`column
 LDA B12
 >>> SMULT
 CLC
 ADC P1T
 STA P1T
 LDA E22
 >>> SMULT
 CLC
 ADC P2T
 STA P2T
 LDA H32
 >>> SMULT
 CLC
 ADC P3T
 STA P3T
 LDY ]3 ;Third`column
 LDA C13
 >>> SMULT
 CLC
 ADC P1T
 STA P1T
 LDA F23
 >>> SMULT
 CLC
```

```
 ADC P2T
 STA P2T
 LDA I33
 >>> SMULT
 CLC
 ADC P3T
 STA ]3 ;Rotated`Z
 TAX
 LDY ZDIV,X ;Table`of`d/(z+z0)
   ;Now`Y`contains`projection`const

 LDA P1T
 >>> SMULTZ
 LDX ZOOM
 CPX #64
 BEQ CONTX
 STY TEMP1
 LDY ZOOM
 >>> SMULT
 LDY TEMP1
CONTX CLC
 ADC #64 ;Offset`the`coordinate
 STA ]1 ;Rotated`and`projected
 CMP LOCXMIN ;See`if`it`is`a`local`minimum
 BCS NOTXMIN
 STA LOCXMIN
NOTXMIN CMP LOCXMAX
 BCC NOTXMAX
 STA LOCXMAX

NOTXMAX LDA P2T
 >>> SMULTZ
 CPX #64
 BEQ CONTY
 LDY ZOOM
 >>> SMULT
CONTY CLC
 ADC #64
 STA ]2 ;Rotated`and`projected`Y
 CMP LOCYMIN
 BCS NOTYMIN
 STA LOCYMIN
NOTYMIN CMP LOCYMAX
 BCC NOTYMAX
 STA LOCYMAX

NOTYMAX <<<  ;All`done

*`LDA`#<EORBUF`;First`we`need`to`clear`the
*`STA`BUFFER`;EOR`buffer
*`LDA`#>EORBUF
*`STA`BUFFER+1

 LDA #0 ;Reset`Ymin`and`Ymax
 STA LOCYMAX
 STA LOCXMAX
 LDA #$FF
 STA LOCYMIN
 STA LOCXMIN

READPTS LDY INDEX
 LDA POLYLIST,Y
 STA P1X
 INY
 LDA POLYLIST,Y
 STA P1Y
 INY
 LDA POLYLIST,Y
 STA P1Z
 INY
 DEC COUNTPTS
 LDA POLYLIST,Y
 STA P2X
 INY
 LDA POLYLIST,Y
 STA P2Y
 INY
 LDA POLYLIST,Y
 STA P2Z
 INY
 DEC COUNTPTS
```

```
 LDA POLYLIST,Y
 STA P3X
 INY
 LDA POLYLIST,Y
 STA P3Y
 INY
 LDA POLYLIST,Y
 STA P3Z
 INY
 STY INDEX
 >>> PROJECT,P1X;P1Y;P1Z
 >>> PROJECT,P2X;P2Y;P2Z
 >>> PROJECT,P3X;P3Y;P3Z

 LDA HIDE
 BEQ :DOIT
 LDA P2X ;Hidden`face`check
 SEC
 SBC P1X
 TAY  ;Y=(x2-x1)
 LDA P3Y
 SEC
 SBC P2Y ;A=(y3-y2)
 >>> SMULT
 STA TEMP1
 LDA P3X
 SEC
 SBC P2X
 TAY
 LDA P2Y
 SEC
 SBC P1Y
 >>> SMULT
 CMP TEMP1 ;If`x1*y2-y1*x2`>`0`then`face
 BMI :DOIT ;is`visible
 DEC COUNTPTS ;Otherwise`read`in`remaining
 BEQ :ABORT ;points`and`return
:POOP INC INDEX
 INC INDEX
 INC INDEX
 DEC COUNTPTS
 BNE :POOP
:ABORT RTS

:DOIT LDA P1X
 STA X1
 LDA P1Y
 STA Y1
 LDA P2X
 STA X2
 LDA P2Y
 STA Y2
 JSR DRAW
 LDA P2X
 STA X1
 LDA P2Y
 STA Y1
 LDA P3X
 STA X2
 LDA P3Y
 STA Y2
 JSR DRAW

 DEC COUNTPTS
 BNE POLYLOOP ;Is`it`just`a`triangle?
 JMP POLYDONE

POLYLOOP LDY INDEX
 LDA POLYLIST,Y
 STA P2X
 INY
 LDA POLYLIST,Y
 STA P2Y
 INY
 LDA POLYLIST,Y
 STA P2Z
 INY
 STY INDEX
 >>> PROJECT,P2X;P2Y;P2Z

 LDA P2X
```

```
 STA X1
 LDA P2Y
 STA Y1
 LDA P3X
 STA X2
 LDA P3Y
 STA Y2
 JSR DRAW

 LDA P2X
 STA P3X
 LDA P2Y
 STA P3Y
 DEC COUNTPTS
 BEQ POLYDONE
 JMP POLYLOOP
POLYDONE LDA P1X ;Close`the`polygon
 STA X2
 LDA P1Y
 STA Y2
 LDA P3X
 STA X1
 LDA P3Y
 STA Y1
 JSR DRAW
 RTS

 TXT 'Same`thing`we`do`every`night,`Pinky:`'
 TXT 'try`to`take`over`the`world!'


*-------------------------------
*`General`questionable-value`error`procedure

*CHOKE`LDX`#00
*:LOOP`LDA`:CTEXT,X
*`BEQ`:DONE
*`JSR`CHROUT
*`INX
*`JMP`:LOOP
*:DONE`RTS
*:CTEXT`HEX`0D`;CR
*`TXT`'something`choked`:('
*`HEX`0D00
*
 TXT 'Narf!'

*-------------------------------
*`Drawin'`a`line.``A`fahn`lahn.

***`Some`useful`macros

CINIT MAC  ;Macro`to`initialize`the`counter
 LDA ]1 ;dx`or`dy
 LSR
 <<<  ;The`dx/2`makes`a`nicer`looking`line

*****`Macro`to`take`a`step`in`X

XSTEP MAC
 LDX DX ;Number`of`loop`iterations
 >>> CINIT,DX
XLOOP LSR CHUNK
 BEQ FIXC ;Update`column
 SBC DY
 BCC FIXY ;Time`to`step`in`Y
 DEX
 BNE XLOOP
DONE LDA OLDX ;Plot`the`last`chunk
 EOR CHUNK
 ORA (BUFFER),Y
 STA (BUFFER),Y
 RTS

FIXC PHA
 LDA OLDX
 ORA (BUFFER),Y ;Plot
 STA (BUFFER),Y
 LDA #$FF ;Update`chunk
 STA OLDX
 STA CHUNK
```

```
 LDA #$80 ;Increase`the`column
 EOR BUFFER
 STA BUFFER
 BNE C2
 INC BUFFER+1
C2
 PLA
 SBC DY
 BCS CONT
 ADC DX
 IF I,]1 ;Do`we`use`INY`or`DEY?
 INY
 ELSE
 DEY
 FIN
CONT DEX
 BNE XLOOP
 JMP DONE

FIXY ADC DX
 PHA
 LDA OLDX
 EOR CHUNK
 ORA (BUFFER),Y
 STA (BUFFER),Y
 LDA CHUNK
 STA OLDX
 PLA
 IF I,]1 ;Update`Y
 INY
 ELSE
 DEY
 FIN
 DEX
 BNE XLOOP
 RTS
 <<<  ;End`of`Macro`xstep

*****`Take`a`step`in`Y

YSTEP MAC
 LDX DY ;Number`of`loop`iterations
 BEQ DONE ;If`dy=0`it's`just`a`point
 >>> CINIT,DY
 SEC
YLOOP PHA
 LDA OLDX
 ORA (BUFFER),Y
 STA (BUFFER),Y
 PLA
 IF I,]1
 INY
 ELSE
 DEY
 FIN
 SBC DX
 BCC FIXX
 DEX
 BNE YLOOP
DONE LDA OLDX
 ORA (BUFFER),Y
 STA (BUFFER),Y
 RTS

FIXX ADC DY
 LSR OLDX
 SEC  ;Important!
 BEQ FIXC
 DEX
 BNE YLOOP
 JMP DONE

FIXC PHA
 LDA #$80
 STA OLDX
 EOR BUFFER
 STA BUFFER
 BNE C2
 INC BUFFER+1
C2 PLA
 DEX
```

```
 BNE YLOOP
 JMP DONE
 <<<  ;End`of`Macro`ystep

*`Take`an`x`step`in`the`EOR`buffer
*`The`sole`change`is`to`use`EOR`instead`of`ORA

EORXSTEP MAC
 LDX DX ;Number`of`loop`iterations
 >>> CINIT,DX
XLOOP LSR CHUNK
 BEQ FIXC ;Update`column
 SBC DY
 BCC FIXY ;Time`to`step`in`Y
 DEX
 BNE XLOOP
DONE LDA OLDX ;Plot`the`last`chunk
 EOR CHUNK
 EOR (BUFFER),Y
 STA (BUFFER),Y
 RTS

FIXC PHA
 LDA OLDX
 EOR (BUFFER),Y ;Plot
 STA (BUFFER),Y
 LDA #$FF ;Update`chunk
 STA OLDX
 STA CHUNK
 LDA #$80 ;Increase`the`column
 EOR BUFFER
 STA BUFFER
 BNE C2
 INC BUFFER+1
C2
 PLA
 SBC DY
 BCS CONT
 ADC DX
 IF I,]1 ;Do`we`use`INY`or`DEY?
 INY
 ELSE
 DEY
 FIN
CONT DEX
 BNE XLOOP
 JMP DONE

FIXY ADC DX
 PHA
 LDA OLDX
 EOR CHUNK
 EOR (BUFFER),Y
 STA (BUFFER),Y
 LDA CHUNK
 STA OLDX
 PLA
 IF I,]1 ;Update`Y
 INY
 ELSE
 DEY
 FIN
 DEX
 BNE XLOOP
 RTS
 <<<  ;End`of`Macro`xstep


*`Take`a`y-step`in`the`EOR-buffer
*`Changes`from`above`are:`only`plot`last`part`of`each
*`vertical`chunk,`don't`plot`last`point,`plot`with`EOR

EORYSTEP MAC
 LDX DY ;Number`of`loop`iterations
 BEQ DONE ;If`dy=0`it's`just`a`point
 >>> CINIT,DY
 SEC
*YLOOP`PHA
*`LDA`OLDX
*`ORA`(BUFFER),Y
*`STA`(BUFFER),Y
```

```
*`PLA
YLOOP IF I,]1
 INY
 ELSE
 DEY
 FIN
 SBC DX
 BCC FIXX
 DEX
 BNE YLOOP
*DONE`LDA`OLDX
*`ORA`(BUFFER),Y
*`STA`(BUFFER),Y
DONE RTS

FIXX ADC DY
 PHA  ;We`only`plot`the`last`part`of`each`chunk
 LDA OLDX
 EOR (BUFFER),Y
 STA (BUFFER),Y
 PLA
 LSR OLDX
 SEC  ;Important!
 BEQ FIXC
 DEX
 BNE YLOOP
 JMP DONE

FIXC PHA
 LDA #$80
 STA OLDX
 EOR BUFFER
 STA BUFFER
 BNE C2
 INC BUFFER+1
C2 PLA
 DEX
 BNE YLOOP
 JMP DONE
 <<<  ;End`of`Macro`ystep
****`Initial`line`setup

**`The`commented`lines`below`are`now`taken`care`of`by`the
**`calling`routine.
*DRAW`>>>`MOVE,TX1;X1``;Move`stuff`into`zero`page
*`>>>`MOVE,TX2;X2``;Where`it`can`be`modified
*`>>>`MOVE,TY1;Y1
*`>>>`MOVE,TY2;Y2

DRAW LDA FILL
 BNE :SETEOR
 >>> SETBUF
 JMP :SETUP
:SETEOR LDA #<EORBUF ;Use`EOR`buffer`instead`of
 STA BUFFER ;display`buffer`for`drawing
 LDA #>EORBUF
 STA BUFFER+1

:SETUP SEC  ;Make`sure`x1<x2
 LDA X2
 SBC X1
 BCS :CONT
 LDA Y2 ;If`not,`swap`P1`and`P2
 LDY Y1
 STA Y1
 STY Y2
 LDA X1
 LDY X2
 STY X1
 STA X2

 SEC
 SBC X1 ;Now`A=dx
:CONT STA DX
 LDX X1 ;Put`x1`into`X,`now`we`can`trash`X1

COLUMN TXA ;Find`the`first`column`for`X
 LSR
 LSR  ;There`are`x1/8`128`byte`blocks
 LSR  ;Which`means`x1/16`256`byte`blocks
 LSR
```

```
 BCC :EVEN ;With`a`possible`extra`128`byte`block
 LDY #$80 ;if`so,`set`the`high`bit
 STY BUFFER
 CLC
:EVEN ADC BUFFER+1 ;Add`in`the`number`of`256`byte`blocks
 STA BUFFER+1

 SEC
 LDA Y2 ;Calculate`dy
 SBC Y1
 BCS :CONT2 ;Is`y2>y1?
 EOR #$FF ;Otherwise`dy=y1-y2
 ADC #$01
:CONT2 STA DY
 CMP DX ;Who's`bigger:`dy`or`dx?
 BCC STEPINX ;If`dx,`then...
 JMP STEPINY

STEPINX LDY Y1
 CPY Y2
 LDA BITP,X ;X`currently`contains`x1
 STA OLDX
 STA CHUNK
 BCC XINCY ;Do`we`step`forwards`or`backwards`in`Y?
 JMP XDECY

XINCY LDA FILL
 BEQ NORMXINC
 >>> EORXSTEP,INY
NORMXINC >>> XSTEP,INY

XDECY LDA FILL
 BEQ NORMXDEC
 >>> EORXSTEP,DEY
NORMXDEC >>> XSTEP,DEY

STEPINY LDY Y1
 LDA BITP,X ;X=x1
 STA OLDX
 LSR  ;Y`doesn't`use`chunks
 EOR OLDX ;So`we`just`want`the`bit
 STA OLDX
 CPY Y2
 BCS YDECY

YINCY LDA FILL
 BEQ NORMINC
 >>> EORYSTEP,INY
NORMINC >>> YSTEP,INY

YDECY LDA FILL
 BEQ NORMDEC
 >>> EORYSTEP,DEY
NORMDEC >>> YSTEP,DEY


*--------------------------------
*`Clean`up

CLEANUP LDA VMCSB ;Switch`char`rom`back`in
 AND #%11110101 ;default
 STA VMCSB

 RTS  ;bye!

 TXT 'spinal`cracker`'
 TXT 'slj`6/95'

*--------------------------------
*`Set`up`bit`table

 DS ^ ;Clear`to`end`of`page
   ;So`that`tables`start`on`a`page`boundary
BITP LUP 16 ;128`Entries`for`X
 DFB %11111111
 DFB %01111111
 DFB %00111111
 DFB %00011111
 DFB %00001111
 DFB %00000111
 DFB %00000011
```

```
 DFB %00000001
 --^

SIN ;Table`of`sines,`120`bytes
COS EQU SIN+128 ;Table`of`cosines
   ;Both`of`these`trig`tables`are
   ;currently`set`up`from`BASIC
ZDIV EQU COS+128 ;Division`table
TMATH1 EQU ZDIV+384 ;Math`table`of`f(x)=x*x/256
TMATH2 EQU TMATH1+512 ;Second`math`table
POLYLIST EQU TMATH2+512 ;List`of`polygons
```

==========

Second SID Chip Installation
Copyright 1988 Mark A. Dickenson

This information and software is COPYRIGHTED and made available on a SHAREWARE basis. This file can be freely copied and distributed as long as it is not SOLD. This information cannot be used to construct and sell a hardware device without receiving prior permission from the author. There is not a set fee for the use of this information. Just send in whatever you feel the information is worth.

If you have any gripes, complaints, suggestions, COMPLIMENTS or DONATIONS of any sort please send them to:

Mark Dickenson
600 South West Street
Nevada, Missouri 64772

Adding an extra SID 6581/6582 chip

This is not a project to be tackled by the sqeamish or people who are deathly afraid of opening their computer just to take a peek inside.

Now let's get rid of the nasty stuff first. No liability is assumed with respect to the use of the following information. In other words if you screw-up trying to install this modification, then it's your responsability.

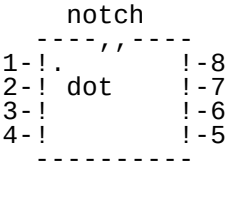
YOU DO THIS AT YOUR OWN RISK!!!!

If you do not feel up to it PLEASE take it to a Commodore repair center or a repair service that can work on computers and let them do the installation. I will warn you that most Commodore Repair Centers will not or do not like to do this modification. When they do, it can be expensive. If you belong to a Users Group, tell them about the project and ask if there is anyone there that could perform the operation. This modification will NOT hurt the computer in any way, unless it is installed WRONG.

You can make your own piggy back board or you can do what I am going to describe (since it is a little hard to put a schematic in a text file).

You should ground yourself with a static guard wristband (such as what Radio Shack sells). Even though the chip is quite durable, just the right static discharge can ruin all or part of the SID chip.

For those of you that are not familier with the way pins are numbered on an IC chip here is a short explanation. On one end of the IC you should find a little notch, looking at the chip with the notch at the top the numbering goes this way. The upper left corner of the chip is pin 1 and they are numbered consecutively, counter-clockwise around the chip. Some chips do not have a notch in one end, but instead dot is placed in one of the chip corners to designate that pin 1 starts in that location.



I have included the information that is needed to install this modification on the Commodore 64, 64C and 128. I haven't been able to look inside the 128D, so I cannot provide the information with any accuracy.

There are TWO different 64C circuit boards and both use DIFFERENT SID chips. You can tell the difference by opening the 64C. If you see a 64-pin chip on the board and the board is only 5.5-6 inches wide then you have the narrow board 64C and must use the 9 volt 6582 SID chip. The number of the

chip in the 64C narrow is an 8520 and is the same as the 6582.

Parts Commodore 64, 64C (wide) & 128

1 - 6581 SID chip from Jamco or Kassara Microsystems
1 - 2N2222 transistor Radio Shack 276-1617
2 - 220pf capacitors Radio Shack 272-124

Parts Commodore 64C Narrow Board

1 - 6582 SID Chip From Jamco or Kassara Microsystems
1 - 2222A transistor Radio Shack 276-2009
2 - .022uf capacitors Radio Shack 272-1066
2 - 1k ohm 1/4 watt resistors Radio Shack 271-1321

Parts 64, 64C (all) & 128

2 - 1k ohm 1/4 watt resistors Radio Shack 271-1321
1 - 1000 pf capacitor Radio Shack 272-126 listed as .001 mf this is
    the same as 1000pf
1 - 10k ohm 1/4 watt resistor Radio Shack 271-1335
1 - 10 uf electrolitic capacitor Radio Shack 272-1025
1 - 5 inch length of wire
1 - 5 inch length of shielded cable
1 - surface mount female RCA plug (this is what you normally find on the back
    of your stereo.

On the C-64 and 64C (wide) the SID is IC U18 (the IC number will be marked in white on the circuit board). It is usually located in the middle of the circuit board, next to the metal video chip case or up between and just below the serial and monitor jacks.

On the C-64C (narrow board) the SID chip is IC U9. It is located in the middle of the board, just a little to the right of center) and called 520.

On the C-128 the SID is IC U5. It is located at the back of the circuit board just to the right of the metal housing for the 40 and 80 column video chips.

First bend out pins 23, 24 and 26 and cut them off of the 6581/6582 SID chip. These are for the two analog and one audio input lines. They will cause problems if connected and since they will not be used it is best to remove them.

Now bend out pins 1, 2, 3, 4, 8, and 27.

Solder one of the 220pf capacitors (64C narrow uses .022 uf) to pins 1 and 2 then solder the other 220pf (64C narrow - .022uf) capacitor to pins 3 and 4. The capacitors control the upper and lower frequency range and filters of the SID chip.

The reason I am using 220pf capacitors is because of problems with the filters in the SID chip. The C-64 first came out with 2200pf capacitors, but they were changed to 470pf. The reason for this was because the filters of the SID vary from chip to chip and using 2200pf caused a lot of them to sound muffeled when the filters were on. I have found that by lowering the capacitor value to 220 pf helps even more. If you wish, you can use 470s if you feel it would be better, but DO NOT use 2200pf.

The 6582 SID chip for the 64C narrow must use the .022uf capacitors, as the filter range is much different.

Solder one end of your wire to pin 8 of the SID chip. This is for the chip select line. We will connect this to the cartridge port. This tells the computer where in memory the chip resides (described later).

Now solder the remaining pins (excluding the ones we have bent out and/or removed 1, 2, 3, 4, 8, 23, 24, 26 and 27) to the sid chip currently in your computer. You may have to bend those pins inward just a little for them to get a good grip on the SID chip. Be very careful not leave the soldering iron on the chip TOO long as you could ruin BOTH SID chips. I would put some heat sink (silicon grease) between the two chips before soldering them together. This will provide better heat dispersal on the bottom chip.

Now that you have the chips soldered together (place the SID chips back in the socket if you removed them), solder the wire from pin 8 (on the SID chip) to pin 7 of the cartridge port on the back of the computer. Set the computer infront of you like to are getting ready to type, with the back of the computer away from you. Look at the cartridge port (located in the upper right corner of the circuit board). You will see two rows of pins connecting the cartridge port to the circuit board. You want the row of pins closest to the front of the computer. Now, count the pins starting at the LEFT side and counting to the right. You want to solder the wire from pin 8 of the extra SID chip to pin number 7 of the cartridge port. This is the same place on all of the models C-64, 64C and 128.

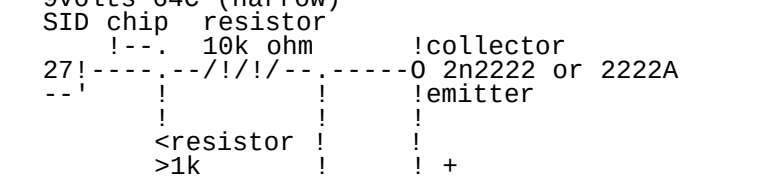
This will tell the computer that the extra SID chip is at address $DE00 hex or 56832 decimal. You will access it just like you would the regular sid chip but starting at this address.

I am no longer describing how to connect for address $DF00. This address causes problems with the RAM Expansion Units and numerous other cartridges. From now on address $DE00 is the ONLY address for the SID chip.

Now partially reassemble your computer (be careful that nothing shorts out the pins still sticking out). Turn the computer on and load the player program provided and tell it to load in 'TEST'. If you get sound then so far so good. Turn off the computer and disassemble the case.

Drill a hole in the back end of the computer just large enough to anchor the RCA plug. Then solder the center wire of the shielded cable to the center post of the RCA plug. Insert the wire through the hole you have just drilled and anchor the plug to the case. Now solder the ground wire to the ground tab on the RCA plug.

Here comes the difficult part to explain. This is the coupling circuit for the audio output. Here is a rough schematic.



==========

SOLVING LARGE SYSTEMS OF LINEAR EQUATIONS ON A C64 WITHOUT MEMORY
by Alan Jones (alan.jones@qcs.org)

OK, now that I have your attention, I lied. You can't solve dense linear systems of equations by direct methods without using memory to store the problem data. However, I'll come back to this memory free assertion later. The main purpose of this article is to rescue a usefull numerical algorithm, "Quartersolve", and also to provide a brief look at the COMAL programming language and BLAS routines.

Linear systems of equations, A(,)*x()=b(), where A is a square matrix and x and b are vectors (or arrays), must often be solved for x in the solution of a variety of problems. The size or dimension of the problem is n and just storing A requires 5*n*n bytes of memory, assuming C64/128 5 byte real variables. The prefered solution method is a form of Gaussian Elimination which requires 1/3 n*n*n multiplications. I'll ignore the additional n*n and n multiplies. For large problems our C64 has two serious limitations, small memory size and slow floating point arithmetic. Problems with n=10 can be computed easily. Problems with n=100 will require 100 times more memory and 1000 times more computing time. The computing time is not a real problem. I don't mind letting my computer run while I watch a movie, sleep, or go on a trip. Calculating or setting up the problem may take much longer than its solution anyway. Available memory is the practical limiting factor. After we use up available RAM we have to resort to other algorithms that will use the disk drive to move data in and out of the computer. The 1541 drive is particularly slow and I would not want to subject it to undue wear and tear.

How big a problem do we need to be able to solve? In many cases the problem itself will fix n and there is no way to reduce it. In other cases you might be modeling a real continuous problem with a discrete number of elements. N should be infinity but for problem solution n=50 might be big enough. Consider calculating the aerodynamic potential flowfield around a body of revolution. You could fix points on the

surface of the body (a meridian) and have a series of sort line segments make up elements to approximate the shape. The lager n is the closer the smooth shape is aproximated and the more accurate the computed solution becomes. n=100 might be a good choice for a simple shape. We could also use a "higher order" menthod. In this case we can substitute a curved line element for the straight line segment. Calculating the matrix elements will be more difficult but n=40 curved elements might give a more accurate solution than 100 flat elements. Another consideration is the resolution of the solution. You might want to plot the solution on the 200 x 320 pixel hi-res C64 screen. 40 points might be too coarse and 320 might be overkill. We might also need to calculate the slope or derivatives from the calculated solution which will require more closely spaced solution points. There are often choices that you can make in modeling a system and selecting a solution algorithm so that a problem can be solved within the limits of a C64. There are often interesting tradeoffs in memory requirements and execution speed.

How big a problem can we solve with a C64? Using Quartersolve with assembly language we can probably do n=200 or more. If we are going to store the problem data on a single 1541 diskette and read it in a row at time we can only do n=182 or so. Actually I think n should be well under 100. Different operating systems and languages limit the amount of useable RAM; BASIC 40K, COMAL 2.0 30K, GEOS 23K, the initial disk loaded COMAL 0.14 10K... Solving a linear system may only be a small subproblem inside a large application program. The idea is to be able to solve reasonable sized problems using your prefered computing environment without having to do a lot of chaining or loading of separate programs. Quartersolve can free up a lot of memory for other routines or allow n to be doubled.

SPEED

There are a few things that we can do to speed up the calculations. First we can select a fast programming language. I prefere COMAL 2.0 which is a fast three pass interpreter. Using an assembler could be the fastest and provide the most useable memory. A true compiler such as C or Pascal could also be a good choice. BASIC is a poor choice except that it is built in and free. In most cases execution can be sped up with some machine language routines like the BLAS (Basic Linear Algebra Subroutines). Calculation speed is measured in FLOPS/sec (Floating Point OPerationS) where, c(i#,j#):=c(i#,j#) + a(i#,k#)*b(k#,j#) is the operation. It is one FP multiply, one FP add, and some indexing overhead. With some interpreters the indexing and interpreting overhead can far exceed the actual FP multiply time. With assembled code the FP multiply time should dominate. I use a ML level 1 BLAS package with COMAL 2.0. For example:

```
c(i#,J#):+sdot(n#,a(i#,1),1,b(1,j#),sdb#)
FOR k#:=1 to n# do c(i#,j#):+a(i#,k#)*b(k#,j#)
```

both calculate the same thing, a dot product with n# FLOPS. For large n# on a C64 the BLAS approach about 320 FLOPS/sec., The overhead of calling the procedure from the interpreter is about the equivalent of 4 FLOPS. Of course modern computer performance is measured in MegaFLOPS/sec. with 8 byte reals (super computers run hundreds or thousands of MFLOPS/sec.). They also inflate the performance by counting the multiply and add as two FLOPS. In his article I use the "old flops" or number of multiplies.

It may also be possible to code 6502 FP arithmetic routines using lookup tables that may perform faster than the built in routines. We could also use the CPU in the disk drives to do distributed processing. But this is beyond the scope of this article.

SOLUTION METHODS

Consider the following choices for numerical solution algorithms:

METHOD	MEMORY	FLOPS
Gaussian Elimination	n*n	1/3 n*n*n
Cholesky Decomposition	1/2 n*n	1/6 n*n*n
QR decomposition	n*n	2/3 n*n*n
QR updating	1/2 n*n	2 n*n*n
Gauss-Jordan	n*n	1/2 n*n*n
Quartersolve	1/4 n*n	1/2 n*n*n

Gaussian Elimination is the prefered method when enough memory is available. In modern terminology this is LU decomposition where A is decomposed or factored into a lower triangular matrix and an upper

triangular matrix. Partial pivoting of rows or columns is an additional complication often required for a stable solution. After the LU decompostion you can readily solve for any number of right hand side vectors in n*n flops each. In addition you can calculate matrix condition number estimates and use iterative improvement techniques. The LU decomposition is done in place overwriting the problem matrix A.

Cholesky Decomposition is a specialized version of Gaussian Elimination for symetric positive definite matrices only. Since A is symetric we only need n*(n+1)/2 memory storage locations. The L and U triangular matrices are simply transposes of the other so only one needs to be stored and is computed in place overwriting the original storage used for A. No pivoting is required. This algorithm cannot solve general nonsymetric problems and is included only for comparison.

QR decomposition factors A into an orthogonal matrix Q and a triangular matrix R. QR decomposition is very stable and can be performed without pivoting. Since Q is orthogonal its inverse is just Q transpose. To solve the linear system we multiply the right hand side vector by Q transpose then solve the triangular system R. Q is computed in a special compact form and stored in the space below R. The decomposition is done in place in the storage used for A, plus an additional n storage locations. QR decomposition requires about twice as many flops as Gaussian Elimination.

There is a variation of the QR solution known as QR updating. The problem is solved a row at a time. A Row of A can be read in from disk storage or calculated as needed. Only R needs to be stored in main memory, n*(n+1)/2 memory locations. R is initialy the identity matrix and is updated as each row of A and its right hand side element are processed. Q is not stored, but the right hand side vector is sequentialy multiplied by Q transpose. After all n rows have been processed the solution is found by simply solving the triangular system R. Since this method only needs half as much memory storage as LU decomposition, we can solve problems 40% larger in a limited memory space. However, the cost in flops is high. Actually QR updating is best used for solving large overdetermined least squares problems.

Gauss-Jordan is a variation of Gaussian Elimination that reduces A to the Identity matrix instead of to LU factors. By applying the same transformations to to the right hand side that reduce A to the identity matrix, the right hand side becomes the solution at completion. Pivoting is requiered. Gauss-Jordan requires about 50% more flops than Gaussian Elimination and most codes use n*n memory storage. Since the LU factors are not computed we can't solve additional right hand side vectors later, or estimate the matrix condition number, or use iterative improvement techniques. It will solve multiple right hand sides that are available from the start.

Quartersolve is a clever implementation of Gauss-Jordan(?) that solves the problem a row at a time like QR updating but only requires 1/4 n*n memory storage. With fixed available memory Quartersolve can solve a problem twice as large as Gaussian Elimination but with a modest performance penalty. Solving a 2n problem with Quartersolve would take 12 times longer (instead of 8) than Gaussian Elimination on a size n problem.

My recommendation is to use Gaussian elimination for solving dense general systems of linear equations when enough main memory is available and switch to Quartersolve for larger problems. For solving huge problems requiering external storage a blocked version of QR decomposition might work best. Cholesky decomposition should be used for symetric positive definite problems. Large problems are often sparse, containing lots of zeros that need not be stored. Specialized code exists for solving many sparce problems, particularly banded matrices, and many of these methods can be used on a C64. Codes for solving unstructured sparce problems are not very suitable for the C64 since they are complex and reduce the amount of memory available for solving the problem. However, large sparce problems can also be solved on the C64 by iterative methods such as Gauss-Siedel and Conjugate Gradient algorithms.

QUARTERSOLVE

Quartersolve is a useful method for solving general dense systems of linear equations that I discovered almost by accident while doing random research in the library. I have not seen any recent texts or papers mentioning this algorithm. I have not seen any reference to it in the C64 literature either. At least one older text mentioned it in passing saying that the code was too long or complex. This is a valid point since usualy the code size directly subtracts from the problem storage.

The code is longer than the Gaussian Elimination code but in my implementation it only takes about 2K of main memory storage and it is a real advantage on the C64. With a C64 we can also put the entire code in an EPROM on a cartridge so the code size is of little concern.

I found Quartersolve in Ref. 1 (R. A. Zamberdino, 1974), which credited the algorithm to Ref. 2 (A. Orden, 1960). I am a little uneasy describing the algorithm since I have not seen Ref. 2 or analyzed the algorithm. I have coded the algorithm, tested it, and used it to solve some large problems on a C64, up to n=90. Zambardino makes two interesting statements in Ref 1. "The number of arithmetic operations is the same as for the Gaussian Elimination method." I am reasonably sure from the description that he meant Gauss-Jordan which requires about 50% more arithmetic than Gaussian Elimination. After processing the ith row only i(n-i) storage locations are required to store the reduced matrix. Max[i(n-i)] = n*n/4. This maximum memory requirement occurs at i = n/2. As i increases further memory required is reduced. Although n*n/4 memory locations must be allocated and dimensioned in an array at the start, Quartersolve always uses the first portion of the array continuously and does not free up memory in holes scattered throughout the array. The C language could possibly use "heap storage" and release the memory for any other use as the procedure advances.

Now back to my initial memory free claim. The large problem that I actually wanted to solve was: A*x=b, B*x=r, for r given b and the square matrices A and B. Elements of A and B are most efficiently calculated at the same time. I could write B to the drive and read it back in after x is computed to calculate r, but I actually wanted to solve this repeatedly inside another loop and I did not want to read and write to a lousy 1541 that much. Using Gaussian elimination would require 2n*n storage. Using Quartersolve could require 1.25n*n storage. However, only n*n storage is needed, that for B. At the ith step the ith row of A and B are calculated. The row of A is processed into the the n*n dimensioned array B filling it from the front. The corresponding row of B is stored in the same array B filling from from the end of array B. As the process continues Quartersolve "dissuses" array B so that rows of B never overwrite storage needed by Quartersolve. At the end we have computed x and all of B is stored in the array B. Simple multiplication produces r. So I can say with pride, at the expense of honesty, that I have solved A*x=b without any additional memory storage for A.

```
PROC slv(n#,nr#,i#,REF a(),REF c(),REF b(,),sdb#,REF sw#(),REF fail#) CLOSED
  // This routine solves a system of equations using the quartersolve
  // algorithm with partial pivoting.
  // It is called a "line at a time" and uses only
  // 0.25*nn memory locations which enables larger problems to be solved.
  // The LU factorization is not available, nor a condition estimate.
  // n# is the dimension of the problem
  // nr# is the number of right hand vectors to be solved for.
  // b(,) is the right hand side columns
  // sdb# is the second dimension of the array b(,)
  USE blas
  USE strings
  q#:=i#-1; m#:=n#-q#; mm1#:=m#-1; fail#:=TRUE; ip1#:=i#+1
  IF i#=1 THEN //initialize pivot array
    FOR j#:=1 TO n# DO sw#(j#):=j#
  ENDIF
  FOR j#:=1 TO q# DO //adjust for previous pivoting
    k#:=sw#(j#)
    WHILE k#<j# DO k#:=sw#(k#)
    IF k#>j# THEN swap'real(c(j#),c(k#))
  ENDFOR j#
  FOR j#:=i# TO n# DO c(j#):-sdot(q#,c(1),1,a(j#-q#),m#)
  p#:=q#+isamax#(m#,c(i#),1)
  r:=ABS(c(p#))
  IF r=0 THEN RETURN
  fail#:=FALSE
  IF p#<>i# THEN
    swap'real(c(i#),c(p#))
    swap'integer(sw#(i#),sw#(p#))
    sswap(q#,a(1),m#,a(p#-q#),m#)
  ENDIF
  r:=1/c(i#)
  IF mm1#<>0 THEN sscal(mm1#,r,c(ip1#),1)
  FOR j#:=1 TO nr# DO b(i#,j#):=r*(b(i#,j#)-sdot(q#,c(1),1,b(1,j#),sdb#))
  FOR k#:=1 TO nr# DO saxpy(q#,-b(i#,k#),a(1),m#,b(1,k#),sdb#)
  IF mm1#>0 THEN
    t#:=1
    FOR j#:=1 TO q# DO
      r:=a(t#); t#:+1
```

```
      scopy(mm1#,a(t#),1,a(t#-j#),1)
      saxpy(mm1#,-r,c(ip1#),1,a(t#-j#),1)
      t#:+mm1#
    ENDFOR j#
    scopy(mm1#,c(ip1#),1,a(mm1#*q#+1),1)
  ELSE //unscramble solution from pivoting
    FOR j#:=1 TO nr# DO
      FOR k#:=1 TO n# DO c(sw#(k#)):=b(k#,j#)
      scopy(n#,c(1),1,b(1,j#),sdb#)
    ENDFOR j#
  ENDIF
ENDPROC slv
//
n#:=8; sdrh#:=1; nrh#:=1; nr#:=1
// a is of  dimension n*n/4
DIM a(16), b(n#), rhs(n#,nrh#), sw#(n#)
FOR i#:=1 TO n# DO
  FOR j#:=1 TO n# DO b(j#):=2297295/(i#+j#-1)
  s:=0
  FOR j#:=n# TO 1 STEP -1 DO s:+b(j#)
  rhs(i#,1):=s
  slv(n#,nr#,i#,a(),b(),rhs(,),sdrh#,sw#(),fail#)
  IF fail# THEN
    PRINT "singularity detected at i=";i#
    STOP
  ENDIF
ENDFOR i#
FOR j#:=1 TO n# DO PRINT rhs(j#,1);
END
```

The Quartersolve algorithm is presented here as a COMAL 2.0 procedure "slv". COMAL is pretty much a dead language and I don't expect anyone to run this code. However, COMAL is a structured algorithmic language that is easy to read. You can readily translate it into the programming language of your choice. Slv is coded as a CLOSED procedure as a personal matter of style. An open procedure would execute faster. The arrays are passed by REFERENCE and do not allocate additional local storage. The main program is just an example for testing. It calls slv n times to solve the linear system. The test problem solves a scaled Hilbert matrix which is ill conditioned. In the absence of roundoff error the solution should be a vector of ones. I usually dimension a() to n#*(n#+1)/4. Slv is presented in its full generality, but you may want to make some simplifications.

Slv can handle multiple right hand side vectors with the two dimensional array b(,) in most applications you will only use a single vector, nr#=1, and you can make some simplifications by just using a one dimensional array.

Pivoting also complicates the code. Problems which are positive definite, or diagonally dominant, or sometimes just well conditioned can be safely solved without pivoting. Stripping out the pivoting code is straight forward and will shorten the code and speed execution.

Anything following // is a comment and can be deleted from your running code.

In COMAL 2.0 you can also "protect" the code which will strip out comments and other information to make a shorter running version.

The remaining discussion will concern COMAL 2.0 and the BLAS.

COMAL 2.0

COMAL 2.0 is an excellent programming language for the C64/128 and I can't describe all of its virtues here. It has one serious limitation. It does not use "continuation" lines, so line length is limited. This is most restrictive in function and procedure lines where it limits the number of parameters that can be passed. Line length is limited to 80 characters. However, if you use a separate text editor or word processor you can enter 120 character lines. Comal will actually execute tokenized lines up to 256 characters so the limitation is really in the editor rather than COMAL. Procedure and variable names can be quite long in Comal, but are kept short because of the line length limitation. "Quartersolve" was shortened to "slv" for this reason.

a:+t is a shorter faster version a:=a+t, and a:-t is a shorter faster version of a:=a-t. This is most usefull when "a" is an array element or an integer.

Comal 2.0 supports ML packages. A package is a collection of functions

or procedures that can be called and executed. A packaged can be ROMMED and stored in EPROM on the Comal 2.0 cartridge. A package can also be loaded from disk and will normally be stored in a RAM location that COMAL does not use for normal programs. LINK "filename" will load and link the ML package to a Comal program. It will stay attached to the program when the program is saved and loaded, unless it is marked as ROMMED. The entire slv procedure could be coded in assembly language and be placed in a package. The slv procedure uses two packages, strings and blas. The command USE packagename makes all of the functions and procedures of the package known. Alternatively, you could place the USE packagename command in the main program and put IMPORT procedurename inside all of the closed procedures that call procedurename.

Slv calls the swap'real and swap'integer proocedures from the strings package. The strings package is a ROMMED package on the Super Chip ROM.

It does exactly what it says, e.g. swap'real(a,b) is the same as: t:=a; a:=b; b:=t.

Slv calls the sdot, isamax#, sswap, sscal, saxpy, and scopy routines from the blas package. The blas package is LINKed to the program, but it could, and should, be placed on EPROM.

Basic Linear Algebra Subroutines, BLAS

The BLAS were originally written for the Fortran language to speed execution and streamline code used for solving linear algebra and other matrix problems. The LINPACK routines, Ref. 3, use the BLAS and are perhaps the best known. The idea is that the BLAS routines will be highly optimized for a particular computer, coded in ML or a High Order Language. Some operating systems even include BLAS like routines. Writing fast efficient programs is then a simple matter of selecting the best solution algorithm and coding it in a manner that makes best use of the blas routines. There are blas routines for single precision, double precision, and complex numbers. The level 1 BLAS perform operations on rows or columns of an array and typicaly do n scalar operations replacing the inner most loop of code. There are also level 2 BLAS that perform n*n operations and Level 3 BLAS that perform n*n*n operations. Nicholas Higham has coded most of the single precision level 1 blas routines and put them in a Comal 2.0 package. The Comal blas package is included on the Packages Library Volume 2 disk. I am not aware of ML blas routines coded for any other C64/128 languages although this is certainly possible and recommended.

The Comal blas routines behave exactly the same way that the Fortran blas routines do except that Fortran can pass the starting address of an array with just "a", while Comal requires "a(1)". The Comal blas will allow you pass an array, by reference, of single or multiple dimensions and start from any position in the array. If you code the blas routines as ordinary Comal routines you have to pass additional parameters and have separate routines for single dimensioned arrays and two dimensional arrays. Note also that Fortran stores two dimensional arrays by columns, and Comal (like many other languages) stores two dimensional arrays by rows. If you translate code between Fortran and Comal using blas routines you will have to change the increment variables.

```
            Fortran                          Comal
    dimension c(n), a(ilda,isda)     DIM c(n#), a(lda#,sda#)
    scopy(n,c,1,a(i,1),ilda)         scopy(n#,c(1),1,a(i#,1),1)
    scopy(n,c,1,a(1,j),1)            scopy(n#,c(1),1,a(1,j#),sda#)
```

The first scopy copies array c into the ith row of array a. The second scopy copies array c into the jth column of array a.

This is what scopy does in Fortran:

```
    subroutine scopy(n,sx,incx,sy,incy)
    real sx(1),sy(1)
    ix=1
    iy=1
    do 10 i = 1,n
      sy(iy) = sx(ix)
      ix = ix + incx
      iy = iy + incy
 10 continue
    return
    end
```

The Comal BLAS does exactly the same thing. If coded entirely in COMAL rather than as a package it would have to be different. The call would

change.

scopy(n#,c(1),1,a(1,j#),sda#) would have to become,
scopy(n#,c(),1,1,a(,),1,j#,sda#,sda#) and the Comal procedure might be:

```
PROC scopy(n#, REF x(), ix#, incx#, REF y(,), iy#, jy#, sdy#, incy#) CLOSED
  iyinc#:=incy# DIV sdy#  //assuming y is dimensioned y(?,sdy#)
  jyinc#:=incy# MOD sdy#
  FOR i#=1 TO n# DO
    y(iy#,jy#):=x(ix#)
    ix#:+incx#; iy#:+iyinc#; jy#:+jyinc#
  ENDFOR
ENDPROC scopy
```

Note that more information has to be passed to the procedure and used that the ML blas picks up automatically. Also we would need separate procedures to handle every combination of single and multi dimensional arrays. The Comal ML blas are indeed wonderful. For speed considerations this should also be left as an open procedure or better yet just use in line code.

Here is a very simplified description of what each of the routines in the Comal BLAS package does.

```
sum:=sasum(n#,x(1),1)  Returns sum of absolute values in x().
  sum:=0
  FOR i#:=1 TO n# DO sum:+ABS(x(i#))

saxpy(n#,sa,x(1),1,y(1),1)  Add a multiple of x() to y().
  FOR i#:=1 TO n# DO y(i#):+sa*x(i#)

prod:=sdot(n#,x(1),1,y(1),1)  Returns dot product of x() and y().
  prod:=0
  FOR i#:=1 TO n# DO prod:+x(i#)*y(i#)

sswap(n#,x(1),1,y(1),1)  Swaps x() and y().
  FOR i#:=1 TO n# DO t:=x(i#); x(i#):=y(i#); y(i#):=t

scopy(n#,x(1),1,y(1),1)  Copy x() to y().
  For i#:=1 TO n# DO y(i#):=x(i#)

max#:=isamax#(n,x(1),1)  Returns index of the element of x() with the
                          largest absolute value.
  t:=0; max#:=1
  FOR i#:=1 TO n#
    IF ABS(x(i#))>t THEN t:=ABS(x(i#)); max#:=i#
  ENDFOR i#

sscal(n#,sa,x(1),1)  Scale x() by a constant sa.
  FOR i#:=1 TO n# DO x(i#):=sa*x(i#)

snrm2(n#,x(1),1)  Returns the 2 norm of x().
  norm2:=0
  FOR i#:=1 TO n# DO norm2:+x(i#)*x(i#)
  norm2:=SQR(norm2)

srot(n#,x(1),1,y(1),1,c,s)  Apply Givens rotation.
  FOR i#:=1 TO n# DO
    t:=c*x(i#) + s*y(i#)
    y(i#):=s*x(i#) + c*y(i#)
    x(i#):=t
  ENDFOR i#
```

Bear in mind that each of these simple examples can be more complex as was given for scopy. You now have enough information to write your own BLAS routines in ML or the programming language of your choice, or to expand the BLAS routine calls in slv to ordinary in line code.

You can also apply the BLAS routines in creative ways besides just operating on rows or columns. For example you could create the identity matrix with:

```
  DIM a(n#,n#)
  a(1,1):=1; a(1,2):=0
  scopy(n#*n#-2,a(1,2),0,a(1,3),1) // zero the rest of the matrix
  scopy(n#-1,a(1,1),0,a(2,2),n#+1) // copy ones to the diagonal.
```

References

1. Zambardino, R. A., "Solutions of Systems of Linear Equations with

Partial Pivoting and Reduced Storage Requirements", The Computer Journal
Vol. 17, No. 4, 1974, pp. 377-378.

2. Orden A., "Matrix Inversion and Related Topics by Direct Methods",
in Mathematical Methods for Digital Computers, Vol. 1, Edited by A.
Ralston and H. Wilf, John Wiley and Sons Inc., 1960.

3. Dongarra, J. J., Moeler, C. B., Bunch, J. R., Stewart, G. W.,
Linpack Users' Guide, SIAM Press, Philadelphia, 1979.

==========

The World of IRC - A New Life for the C64/128

by Bill Lueck (coolhand on IRC)

1) Introduction

With the mysterious and magnificent world of the Internet growing at an astounding rate - like doubling every year - readers of this magazine should find that the Internet is actually available to them now - or at least very soon. In fact, most readers of C= Hacking probably get there copies of this magazine on the Internet.

The Internet is not simple. It has complexities and intricacies that can baffle the most erudite and experienced computer scientists in the world. But, for the purposes of this article, maybe you can just accept that the Internet is a worldwide connection of data lines that let computers all over the world talk to each other.. and more importantly, that allow the PEOPLE using the computers all over the world to talk to other computers.. and to talk to other PEOPLE! Here, then, lies the foundation for IRC: it is the mechanism on the Internet that allows PEOPLE to talk to other PEOPLE.

2) Getting on the Net

If you obtained this magazine via the Internet, then you have passed Step 1 (finding a site)! If you do not have access to the Internet (and have not tried), then you need to look around. Possible sites may be a college/university, your employer (use with care), or a commercial provider.

If you are enrolled in college, then you probably have an account, or you may be entitled to one, with no or little cost. The policies on student accounts vary a lot from institution to institution, and from country to country. But check into it.. it is one of the most common methods of Internet access.

If you are employed, and your company has access to the Internet, it may be possible for you to use their facilities. Just a word of caution - make sure that it is ok with your employer to use his facilities... and not on "company time".

Another way that is becoming increasingly more common is to use commercial "Internet providers". These are companies whose sole purpose is to offer you an "account" and give you access to the Internet. The cost, time on line, storage, access, etc., can vary greatly.. you must shop around a bit.. if you have this choice at all.. for the best deal.

These commercial sites are not always easy to find. There may be several commercial providers in an area, but, strangely, they tend not to advertise. Word-of-mouth through friends, BBSs, or User Groups seem to be the best way to locate the site possibilities. But they CAN provide a very good solution.

Another variation on commercial sites are national companies such as Compuserve, Genie, America Online and Delphi. They provide varying degrees of access.. and possibly at somewhat higher costs than local providers. But, again, it is another option.

There is MUCH to do on the Internet, once you have access to it: telnet, ftp, usenet, archie, gopher, www... These may be just names to you now.. but the are all fascinating parts of the Internet. But this article is intended as an introduction to IRC - a fabulous Internet resource which allows users who have access to a client program called IRCII (most often invoked as "irc") to talk to each other (and often to exchange files) in world-wide conversational "channels" (like "party lines", often called "rooms" on some BBSs). Why is this important to readers of this magazine? Well, there is a channel for c64/128 users on IRC called #c-64, a place where c64/128 users are able to meet and exchange all sorts of information, opinions, and files. More on this later.

3) The IRC Client

First, to use IRC it is necessary to have access to an IRC client. A client is a program, usually available on your local site, which actually interprets and responds to your commands, accepts your typing, and shows you the conversation on the channel(s) you have joined.

The most common way to access IRC from a site is to use the IRCII client that your site makes available. This is most often done by simply typing "irc" at your prompt or invoking the irc option from your menu if you don't have a shell account. The first thing you will notice is that your client is attempting to connect to a "server". A server is a special program, run only on certain sites, that actually provides the backbone of the IRC network.

Most sites have several servers pre-defined. You should see the client trying one or more servers until it connects with one.

With Unix irc clients you can define your own unique set of servers by starting IRC with:

irc nick server1 server2 ... serverN

where "serverX" is the alpha or numeric IP address of the server. This will automatically set your irc nick (handle) and will establish a series of servers that your client will switch to if your connection to IRC gets broken (or if a server is not available when you invoke "irc").

What is an IP address, you ask? Well, a basic premise of the Internet is that each computer on the net (at all sites) has a UNIQUE address - a computer code - that allows other computers to send specific data just to that computer. In that way, computers can make sure that the messages and data files that they want (and YOU want) to send to certain places get to their proper destinations.

IP addresses may be used in an alphabetic or numeric form. In most cases they can be used interchangeably. So, all irc servers have a unique alphabetic (and an equivalent numeric) IP address.

Once an IRC session is in progress, Unix users can change servers by typing:/server newserver where "newserver" is as above, the alpha or numeric IP address of the server you want to switch to. More on servers later; but just to mention few now: irc.indiana.edu (midwest); irc.virginia.edu (east); irc.ctr.columbia.edu (east); irc.math.byu.edu (west); irc.colorado.edu (midwest); irc.texas.net (southwest). There are dozens more. Just ask someone on IRC...or do a few /whois nick commands. You will spot many more.

If your site does not have an irc client, it should be possible to install one yourself. This means that you need to ftp the source code for an irc client to your account on your site, make some usually minor edits, then compile the code in your home directory or a subdirectory below it.

One good site for obtaining the necessary irc client code is cs.bu.edu. cd to /irc/clients. Unix users will find the IRCII client source code in two forms: IRC2.2.9.tar.Z (Unix tar and compress at 471k) or IRC2.2.9.tar.gz (Unix tar and GNU compress at 306k). Both files are the same (except for the compression). Be sure to use "BINARY" mode for the ftp transfer.

Move the file to its own subdirectory if you have not ftp'd it to one already. Then uncompress and untar the file. You should now find a small subdirectory tree of files. Be sure and read the INSTALL file in the top subdirectory.

Also in the main subdirectory, there should be two files that need editing to make the client work with you site. One is "Makefile". In it there are at least two edits. Make INSTALL_EXECUTABLE the path name that u want the executable to reside in. This is most often your home directory or the "bin" subdirectory under your home directory. The other is IRCII_LIBRARY. Set this to the top subdirectory where the IRCII code resides. You also must read through the computer system options and set them for the type of computer and Operating System that your site uses.

The other file is "config.h". Change the #define DEFAULT_SERVER line to the alpha or numeric addy of your primary default server. Be sure to enclose the server in quotes ("server").

For VMS users, there is a subdirectory in "clients" named vms. cd to it. There are two versions - irc176 and ircII-for-vms. The first is a more native VMS version, the second is a Unix-like version. They are both executables, and should run on VMS systems. Try both.. see which you like best.

Another fairly new area of IRC clients is the personal client, running on your own computer which would be connected to the Internet through a version of SLIP or PPP, protocols that move much of the overhead of a normal Internet provider down to your own machine. There are IRC clients available for the PC, Amiga, MAC... and even the rumor of one to be produced for the c64/128. This type of client is expanding very rapidly and will be a significant option for an ever increasing number of Internet users.

If you have Telnet only access from your site, there are some sites which offer a "public" irc client, ones which you can use without having an account at that site.. sorta like anonymous ftp for those of you who know what that is. There are drawbacks, though. There are not many of these public clients, they are often slow in response time, you cannot exchange files with other users (DCC), and many of the sites are not always up. Still, it is one possibility that might work for certain situations. Actually, it is the way that I started on IRC and used it for several months (my site did not have a local client, and I did not know how to install one myself).

The public IRC sites I know about now are tiger.itc.univie.ac.at 6668, sci.dixie.edu 6667, irc.nsysu.edu.tw, and irc.demon.co.uk. They are not available from all sites, and usage is limited. But try them if you need to.

Another variation of the "public" options is to apply for a free Unix account at nyx.cs.du.edu. You will have to be validated, which involves a little paperwork. But once completed, you will have a FREE Unix account with full IRC privileges, including DCC file exchange. Of course, you need a "local" account somewhere with telnet and ftp privileges, but this is often easier to obtain than an account with all options locally.

4) Basics of IRC

Well, hopefully, you will now have an Internet site with a method of accessing IRC. Next, we want to give some tips on using and enjoying IRC and introduce DCC, the command for transferring files between people on IRC... and between "bots" and people.

A "bot", you say? Some of you may laugh; sure of course, a bot. What else is new? But... I remember that it was ages before I finally figured out.. or someone gave me a clue.. as to what a "bot" really was. Before we go on, let me give you a VERY brief description of a bot. We can say that a bot may be a "script", a series of IRC language statements understood by your IRC client; or it may be a separate program (typically written in "C"); which, in either case, runs without any help from its "owner" - YOU.

Instead, a bot is intended to respond to others on IRC who "talk" to it by "/msg", "/dcc chat", or even "on-channel" commands like "!list" or "&help". One bot even lists the c64 files it has on-line in response to someone typing "load "$",8".

What a bot does and how you command it varies a LOT. There really is no standard way to talk to a bot. Try "/msg <botname>" help as a starting point and see what happens. Most often there will be instructions that tell you what to do next. Experiment a little - you will get the hang of it.

Back to the main plot. The first thing to do after you get connected to IRC is to choose a "nick". This is the handle that you will be known by and talked to on IRC. Do this by typing:

/nick <nick>

It's your choice.. unless someone else is already using it. IRC does not let two people use the same nick at the same time. It will tell you about this if you try - sometimes in a rather active way - like "kick" you off. Don't worry - just reconnect.. but try a different nick. Try just changing the nick a little - like even putting a "1" or "2" behind it.

Any number of people, however, can use the same nick at different times.

This CAN cause a little confusion.. make sure you know you are talking to who you think you are.. check a nick's whole address with:

/whois <nick>

Next, you will want to join a channel. Do this by typing:

/join #<channel>

A channel is a logical connection of all IRC users anywhere in the world that have typed the same /join command. All lines typed to the channel by anyone on the channel are spread by IRC to all other people who are on the channel. This is the real power of IRC... a world-wide "conference" or "party line", where people with the same interests can communicate with each other.

Because of different delays in different parts of the Internet, all the lines typed by everyone will not always appear at the same time or even in the same order at everyone's terminal. This usually does not cause much of a problem - just be aware that it happens.

If the channel name does not exist at the time you type /join, it will be created for you! Yes, anyone can "create" a channel. But #c-64 is almost always there. Give it a try!

After you get on a channel, you can type:

/who *

This will give you a list of who (which nicks) is on the channel and what their home sites are. This address may or may not be the correct email address for the nick - so check with the person first (perhaps a "/msg <nick>" - see below) if you want to email him.

As mentioned before, normal channel conversation is seen by everyone who has joined the channel. This is great most of the time. Occasionally, though, you may want to tell just one person (or bot) something that the entire channel would not want to hear. In this case, use the command:

/msg <nick> <message>

Type it on a line of its own, and just <nick> will see your <message>. Quite handy for the more "personal" or "specialized" conversations. Careful, though... use the wrong <nick> or leave out the "/" and people other than you intended will see your <message>.

If you find you are doing a lot of /msg's to the same nick, try:

/query <nick>

This will put you in a sort of 'permanent' /msg <nick> mode, so that everything you type that would normally go to the channel will not act like a "/msg <nick>" preceded it, and it will go just to <nick>. Type just "/query" to cancel this mode.

Let's jump, now, to /dcc, the command that allows most IRC users to transfer files. DCC stands for "Direct Client to Client". What it does is allow two nicks to transfer files *directly* between their sites, not going through either of their servers. One of the nicks can even be a bot; IRC does not make a distinction.

When two nicks exchange files, the sender must always start by typing:

/dcc send <nick> <filename>

The recipient will get a message telling what file is being offered and must type:

/dcc get <nick> [filename]

The [filename] is optional, but must be used if more than one file is to be transferred simultaneously. Yes, simultaneous transfer of multiple files CAN be done. Many people do not realize this. Just use the [filename] option with the "/dcc get" command.

The files that you send and the files that you receive with DCC are always in the directory you are in when you start IRC. You can type "/cd" to see what that directory is and you can type:

/cd <pathname>

to change that directory. Or, you can give the absolute or relative pathname of the file you want to send if it is not in your "local" directory.

There are often a couple of bots on #c-64 that can give you c-64 files. "coolhand" is partly a script bot that currently has a lot of c-64 files available for DCC. If coolhand is on IRC, type:

/msg coolhand xdcc list

to see a list of lists (of files). To see the individual files on list n, type:

/msg coolhand xdcc list #n

To have coolhand's script dcc you file #n, type:

/msg coolhand xdcc send #n

followed by:

/dcc get coolhand

when you get the dcc offer message.

There are many scripts that you can use that will autoget a file that is DCC'd to you. The xdcc script that coolhand uses is one such script. (Yes, coolhand will also autoget a file that you send to it.)

5) What/Who is on IRC?

Ok, now you are on IRC. So what will you find? Who is on the #c-64? The answers are quite varied.. and constantly changing. I personally have been on IRC for over 2 years.. (or is it 3?) And I have yet to ascertain an absolute pattern of people or topics. Frustrating? Well, maybe to some. But interesting? Yes, most certainly. IRC, and the #c-64 channel, is a microcosm of the world, with all its variety of people, personalities, projects, propaganda, and priorities. It is a capability, a tool for communications, that is unexcelled in its scope and possibilities.

IRC is totally international, and so is the #c-64 channel. Besides the U.S. and Canada, Europe is very well represented. There is also a smaller but increasingly active contingent in Australia, as the net becomes more accessible there. You will also find a few c-64 users in S. America, Africa, and Asia. Russia and other former Soviet Union countries also have a presence. English is the accepted language for use on #c-64, although you will occasionally see a few other language used for brief times.

What is the channel used for? Just about anything you can imagine that normal conversation would be used for. With a special emphasis for the special interest of most channel participants - the c64/128. For the most part, almost everyone on the channel has had or still has a c64 or c128. Some are active users on a real c64/128, while others use one of the several emulators that exist for various platforms. Many former and current 64 "scene" members are finding their way to the channel, but all members of the c64 community are always welcome, and all are treated equally.

Many people find IRC and #c-64 a very useful way to exchange information quickly without having to wait for email to pass back and forth. As was mentioned before, the DCC capability allows for immediate transfer of files, another quick and effective way to pass information and things like utilities and coding examples. Such capabilities have encouraged many people to either return to the c64 or take up using and programming it for the first time. Yes, the c64 community is actually growing again, thanks in part to the growing presence of the Internet, IRC, and #c-64!

So, when you first get on the channel for the first time, don't be afraid to ask for help. You will probably find that the people on there are either new themselves, or were once new at one time and had the same uncertainties and questions that you do. Most everyone is very willing to help new people. So ask. Also, if you have knowledge or a talent to offer or a willingness to help somehow, just make that known. The channel is full of people, some of whom probably need exactly what you have to give.

A key thing: be patient! When you are new on the channel, you may not be noticed right away, especially if there are several conversations

already going on. In other cases, you may find that there is really no one on the channel, except maybe a few bots. So hang in there or come back a bit later. Believe me, the IS a lot of action on #c-64 most of the time.

Besides being patient yourself, be patient with other people on the channel. Like in the non-cyber world, misunderstandings CAN occur, since your total communication with other people is via the typed word. But the same rules of courteousness that common society utilizes also apply on IRC. Treat people with respect and kindness, and they will most likely respond in a like manner. Sounds like the golden rule? I think so, and I think you will find that its works pretty well on IRC as it does in other life.

Hopefully, this article will help you get started enjoying IRC and particularly #c-64. There's a lot to be gained there... information, files, and even new friends. It's a way to give our c64 community new life and spirit. Give it a try! See you there.

****

Some of the material in this article was previously published in "Driven" and is used here by permission.

==========
SwiftLink-232 Application Notes (version 1.0b)

This information is made available from a paper document published by CMD, with CMD's express written permission. [This version includes a couple of grammatical corrections and minor changes, plus, the source code has been debugged and extended by Craig Bruce <csbruce@ccnga.uwaterloo.ca>.]

1. INTRODUCTION

The SwiftLink-232 ACIA cartridge replaces the Commodore Kernal RS-232 routines with a hardware chip. The chip handles all the bit-level processing now done in software by the Commodore Kernal. The ACIA may be accessed by polling certain memory locations in the I/O block ($D000 - $DFFF) or through interrupts. The ACIA may be programmed to generate interrupts in the following situations:

1) when a byte of data is received
2) when a byte of data may be transmitted (i.e., the data register is empty)
3) both (1) and (2)
4) never

The sample code below sets up the ACIA to generate an interrupt each time a byte of data is received. For transmitting, two techniques are shown. The first technique consists of an interrupt handler which enables transmit interrupts when there are bytes ready to be sent from a transmit buffer. There is a separate routine given that manages the transmit buffer. In the second technique, which can be found at the very end of the sample code, neither a transmit buffer or transmit interrupts are used. Instead, bytes of data are sent to the ACIA directly as they are generated by the terminal program.

NOTE: The ACIA will _always_ generate an interrupt when a change of state occurs on either the DCD or DSR line (unless the lines are not connected in the device's cable).

The 6551 ACIA was chosen for several reasons, including the low cost and compatibility with other Commodore (MOS) integrated circuits. Commodore used the 6551 as a model for the Kernal software. Control, Command, and Status registers in the Kernal routines partially mimic their hardware counterparts in the ACIA.

NOTE: If you're using the Kernal software registers in your program, be sure to review the enclosed 6551 data sheet carefully. Several of the hardware-register locations do _not_ perform the same function as their software counterparts. You may need to make a few changes in your program to accommodate the differences.

2. BUFFERS

Bytes received are placed in "circular" or "ring" buffers by the sample routine below, and also by the first sample transmit routine. To keep things similar to the Kernal RS-232 implementation, we've shown 256-byte buffers. You may want to use larger buffers for file transfers or to allow more screen-processing time. Bypassing the Kernal routines free many zero-page locations, which could improve performance of pointers to large buffers.

If your program already directly manipulates the Kernal RS-232 buffers, you'll find it very easy to adapt to the ACIA. If you use calls to the Kernal RS-232 file routines instead, you'll need to implement lower-level code to get and store buffer data.

Briefly, each buffer has a "head" and "tail" pointer. The head points to the next byte to be removed from the buffer. The tail points to the next free location in which to store a byte. If the head and tail both point to the same location, the buffer is empty. If (tail+1)==head, the buffer is full.

The interrupt handler described below will place received bytes at the tail of the receive buffer. Your program should monitor the buffer, either by comparing the head and tail pointers (as the Commodore Kernal routines do), or by maintaining a byte count through the interrupt handler (as the attached sample does). When bytes are available, your program can process them, move the head pointer to the next character, and decrement the counter if you use one.

You should send a "Ctrl-S" (ASCII 19) to the host when the buffer is nearing capacity. At higher baud rates, this "maximum size" point may need to be lowered. We found 50 to 100 bytes worked fairly well at 9600 baud. You can probably do things more efficiently (we were using a _very_ rough implementation) and set a higher maximum size. At some "maximum size", a "Ctrl-Q" (ASCII 17) can be sent to the host to resume transmission.

To transmit a byte using the logic of the first transmit routine below, first make sure that the transmit buffer isn't full. Then store the byte at the tail of the transmit buffer, point the tail to the next available location, and increment the transmit buffer counter (if used).

The 6551 transmit interrupt occurs when the transmit register is empty and available for transmitting another byte. Unless there are bytes to transmit, this creates unnecessary interrupts and wastes a lot of time. So, when the last byte is removed from the buffer, the interrupt handler in the first transmit routine below disables transmit interrupts.

Your program's code that stuffs new bytes into the transmit buffer must re-enable transmit interrupts, or your bytes may never be sent. A model for a main code routine for placing bytes into the transmit buffer follows the sample interrupt handler.

Using a transmit buffer allows your main program to perform other takes while the NMI interrupt routine takes care of sending bytes to the ACIA. If the buffer has more than a few characters, however, you may find that most of the processor time is spent servicing the interrupt. Since the ACIA generates NMI interrupts, you can't "mask" them from the processor, and you may have timing difficulties in your program.

One solution is to eliminate the transmit buffer completely. Your program can decide when to send each byte and perform any other necessary tasks in between bytes as needed. A model for the main-code routine for transmitting bytes without a transmit buffer is also shown following the sample interrupt-handler code. Feedback from developers to date is that many of them have better luck _not_ using transmit interrupts or a transmit buffer.

Although it's possible to eliminate the receive buffer also, we strongly advise that you don't. The host computer, not your program, decides when a new byte will arrive. Polling the ACIA for received bytes instead of using an interrupt-driven buffer just waste's your program's time and risks missing data.

For a thorough discussion of the use of buffers, the Kernal RS-232 routines, and the Commodore NMI handler, see "COMPUTE!'s VIC-20 and Commodore 64 Tool Kit: Kernal", by Dan Heeb (COMPUTE! Books) and "What's Really Inside the Commodore 64", by Milton Bathurst (distributed in the US by Schnedler Systems).

3. ACIA REGISTERS

The four ACIA registers are explained in detail in the enclosed data sheets. The default location for them in our cartridge is address $DE00--$DE03 (56832--56836).

3.1. DATA REGISTER ($DE00)

This register has dual functionality: it is used to receive and transmit all data bytes (i.e., it is a read/write register).

Data received by the ACIA is placed in this register. If receive interrupts are enabled, an interrupt will be generated when all bits for a received byte have been assembled and the byte is ready to read.

Transmit interrupts, if enabled, are generated when this register is empty (available for transmitting). A byte to be transmitted can be placed in this register.

3.2. STATUS REGISTER ($DE01)

This register has dual functionality: it shows the various ACIA settings when read, but when written to (data = anything [i.e., don't care]), this register triggers a reset of the chip.

As the enclosed data sheet shows, the ACIA uses bits in this register to indicate data flow and errors.

If the ACIA generates an interrupt, bit #7 is set. There are four possible sources of interrupts:

1) receive (if programmed)
2) transmit (if programmed)
3) if a connected device changes the state of the DCD line
4) if a connected device changes the state of the DSR line

Some programmers have reported problems with using bit #7 to verify ACIA interrupts. At 9600 bps and higher, the ACIA generates interrupts properly, and bits #3--#6 (described below) are set to reflect the cause of the interrupt, as they should. But, bit #7 is not consistently set. At speeds under 9600 bps, bit #7 seems to work as designed. To avoid any difficulties, the sample code below ignores bit #7 and tests the four interrupt source bits directly.

Bit #5 indicates the status of the DSR line connected to the RS-232 device (modem, printer, etc.), while bit #6 indicates the status of the DCD line. NOTE: The function of these two bits is _reversed_ from the standard implementation. Unlike many ACIAs, the 6551 was designed to use the DCD (Data Carrier Detect) signal from the modem to activate the receiver section of the chip. If DCD is inactive (no carrier detected), the modem messages and echos of commands would not appear on the user's screen. We wanted the receiver active at all times. We also wanted to give the you access to the DCD signal from the modem. So, we exchanged the DCD and DSR signals at the ACIA. Both lines are pulled active internally by the cartridge if left unconnected by the user (i.e., in an null-modem cable possibility).

Bit #4 is set if the transmit register is empty. Your program must monitor this bit and not write to the data register until the bit i sset (see the sample XMIT code below).

Bit #3 is set if the receive register is full.

Bits #2, #1, and #0, when set, indicate overrun, framing, and parity errors in received bytes. The next data byte received erases the error information for the preceding byte. If you wish to use these bits, store them for processing by your program. The sample code below does not implement any error checking, but the Kernal software routines do, so adding features to your code might be a good idea.

3.3. COMMAND REGISTER ($DE02)

The Command Register control parity checking, echo mode, and transmit/receive interrupts. It is a read/write register, but reading the register simply tells you what the settings of the various parameters are.

You use bits #7, #6, and #5 to choose the parity checking desired.

Bit #4 should normally be cleared (i.e., no echo)

Bits #3 and #2 should reflect whether or not you are using transmit interrupts, and if so, what kind. In the first sample transmit routine below, bit #3 is set and bit #2 is cleared to disable transmit interrupts (with RTS low [active]) on startup. However, when a byte is placed in the transmit buffer, bit #3 is cleared and bit #2 is set to enable transmit interrupts (with RTS low). When all bytes in the buffer have been transmitted, the interrupt handler disables transmit interrupts. NOTE: If you are connected to a RS-232 device that uses CTS/RTS handshaking, you can tell the device to stop temporarily by bringing RTS high (inactive): clear both bits #2 and #3.

Bit #1 should reflect whether or not you are using receive interrupts. In the sample code below, it is set to enable receive interrupts.

Bit #0 acts as a "master control switch" for all interrupts on the chip itself. It _must_ be set to enable any interrupts -- if it is cleared, all interrupts are turned off and the receiver section of the chip is disabled.

This bit also pulls the DTR line low to enable communication with the connected RS-232 device. Clearing this bit causes most Hayes-compatible modems to hang up (by bringing DTR high). This bit should be cleared when a session is over and the user exits the terminal program to insure that no spurious interrupts are generated. One fairly elegant way to do this is to perform a software reset of the chip (writing any value to the Status register).

NOTE: In the figures on the 6551 data sheet, there are small charts at the bottom of each of the labelled "Hardware Reset/Program Reset". These charts indicate what values the bits of these registers contain after a hardware reset (like toggling the computer's power) and a program reset (a write to the Status register).

3.4. CONTROL REGISTER ($DE03)

You use this register to control the number of stop bits, the word length, switch on the internal baud-rate generator, and set the baud rate. It is a read/write register, but reading the register simply tells you what the various parameters are. See the figure in the data sheet for a complete list of parameters.

Be sure that bit #4, the "clock source" bit, is always set to use the on-chip crystal-controlled baud-rate generator.

You use the other bits to choose the baud rate, word length, and number of stop bits. Note that our cartridge uses a double-speed crystal, so values given on the data sheet are doubled [this is how they are shown below] (the minimum speed is 100 bps and the maximum speed is 38,400 bps).

4. ACIA HARDWARE INTERFACING

The ACIA is mounted on a circuit board designed to plug into the expansion (cartridge) port. The board is housed in a cartridge shell with a male DB-9 connector at the rear. The "IBM(R) PC/AT(TM) standard" DB-9 RS-232 pinout is implemented. Commercial DB-9 to DB-25 patch cords are readily available, and are sold by us as well.

Eight of the nine lines from the AT serial port are implemented: TxD, RxD, DTR, DSR, RTS, CTS, DCD, & GND. RI (Ring Indicator) is not implemented because the 6551 does not have a pin to handle it. CTS and RTS are not normally used by 2400 bps or slower Hayes-compatible modems, but these lines are being used by several newer, faster modems (MNP modems in particular). Note that although CTS is connected to the 6551, there is no way to monitor what state it is -- the value does not appear in any register. The 6551 handles CTS automatically: if it is pulled high (inactive) by the connected RS-232 device, the 6551 stops transmitting (clears the "transmit data register empty" bit [#4] in the status register).

The output signals are standard RS-232 level compatible. We've tested units with several commercial modems and with various computers using null-modem cables up to 38,400 bps without difficulties. In addition, there are pull-up resistors on three of the four input lines (DCD, DSR, CTS) so that if these pins are not connected in a cable, those three lines will pull to the active state. For example, if you happen to use a cable that is missing the DCD line, the pull-up resistor will pull the line active, so that bit #6 in the status register would be cleared (DCD is active low).

An on-board crystal provides the baud rate clock signal, with a maximum of 38.4 Kbaud, because we are using a double-speed crystal. If possible, test your program at 38.4 Kbaud as well as lower baud rates. Users may find this helpful for local file transfers using the C-64/C-128 as a dumb terminal on larger systems. And, after all, low-cost 28.8 Kb modems for the masses are just around the corner.

Default decoding for the ACIA addresses is done by the I/O #1 line (pin 7) on the cartridge port. This line is infrequently used on either the C-64 or C-128 and should allow compatibility with most other cartridge products, including the REU. The circuit board also has pads for users with special needs to change the decoding to I/O #2 (pin 10). This change moves the ACIA base address to $DF00, making it incompatible with the REU.

C-128 users may also elect to decode the ACIA at $D700 (this is a SID-chip mirror on the C-64). Since a $D700 decoding line is not available at the expansion port, the user would need to run a clip lead into the computer and connect to pin 12 of U2 (a 74LS138). We have tried this and it works. $D700 is an especially attractive location for C-128 BBS authors, because putting the SwiftLink there will free up the other two memory slots for devices that many BBS sysops use: IEEE and hard-drive interfaces.

Although we anticipate relatively few people changing ACIA decoding, you

should allow your software to work with a SwiftLink at any of the three locations. You could either (1) test for the ACIA automatically by writing a value to the control register and then attempting to read it back or (2) provide a user-configurable switch/poke/menu option.

The Z80 CPU used for CP/M mode in the C-128 is not connected to the NMI line, which poses a problem since the cleanest software interface for C-64/C-128-mode programming is with this interrupt. We have added a switch to allow the ACIA interrupt to be connected to either NMI or IRQ, which the Z80 does use. The user can move this switch without opening the cartridge.

5. SAMPLE CODE

This section has been translated into ACE-assembler format. Cut on the dotted lines to extract the code, and assemble it using the ACE assembler (ACE is a public-domain program). This program will work on both the C64 and C128. To use from BASIC:

```
LOAD"SAMPLE",8,1
SYS8192
```

It is a very simple terminal program. Press the STOP key to exit from it.

```
%%%---8<---cut-here---8<---%%%
;Sample NMI interrupt handler for 6551 ACIA on Commodore 64/128

;(c) 1990 by Noel Nyman, Kent Sullivan, Brian Minugh,
;Geoduck Development Systems, and Dr. Evil Labs.

;    ---=== EQUATES ===---

base      =    $DE00    ;base ACIA address
data      =    base
status    =    base+1
command   =    base+2
control   =    base+3

;Using the ACIA frees many addresses in zero page normally used by
;Kernel RS-232 routines.  The addresses for the buffer pointers were
;chosen arbitrarily.  The buffer vector addresses are those used by
;the Kernal routines.

rhead     =   $A7      ;pointer to next byte to be removed from
                       ;receive buffer
rtail     =   $A8      ;pointer to location to store next byte received
rbuff     =   $F7      ;receive-buffer vector

thead     =   $A9      ;pointer to next byte to be removed from
                       ;transmit buffer
ttail     =   $AA      ;pointer to location to store next byte
                       ;in transmit buffer
tbuff     =   $F9      ;transmit buffer

xmitcount =   $AB      ;count of bytes remaining in transmit (xmit) buffer
recvcount =   $B4      ;count of bytes remaining in receive buffer

errors    =   $B5      ;DSR, DCD, and received data errors information

xmiton    =   $B6      ;storage location for model of command register
                       ;which turn both receive and transmit interrupts on
xmitoff   =   $BD      ;storage location for model of command register
                       ;which turns the receive interrupt on and the
                       ;transmit interrupts off

NMINV     =   $0318    ;Commodore Non-Maskable Interrupt vector
OLDVEC    =   $03fe    ;innocuous location to store old NMI vector (two bytes)

;    ---=== INITIALIZATION ===---

;Call the following code as part of system initialization.

;clear all buffer pointers, buffer counters, and errors location

      org   $2000      ;change to suit your needs
      lda   #$00
      sta   rhead
      sta   rtail
      sta   thead
      sta   ttail

      sta   xmitcount
```

```
      sta   recvcount
      sta   errors

;store the addresses of the buffers in the zero-page vectors

      lda   #<TRANSMIT_BUFFER
      sta   tbuff
      lda   #>TRANSMIT_BUFFER
      sta   tbuff + 1

      lda   #<RECEIVE_BUFFER
      sta   rbuff
      lda   #>RECEIVE_BUFFER
      sta   rbuff + 1

;the next four instructions initialize the ACIA to arbitrary values.
;These could be program defaults, or replaced by code that picks up
;the user's requirements for baud rate, parity, etc.

;The ACIA "control" register controls stop bits, word length, the
;choice of internal or external baud-rate generator, and the baud
;rate when the internal generator is used.  The value below sets the
;ACIA for one stop bit, eight-bit word length, and 4800 baud using the
;internal generator.
;             .------------------------- 0 = one stop bit
;             :
;             :.-------------------- word length, bits 6-7
;             ::.------------------- 00 = eight-bit word
;             :::
;             :::.-------------- clock source, 1 = internal generator
;             ::::
;             :::: .----- baud
;             :::: :.---- rate
;             :::: ::.--- bits   ;1010 == 4800 baud, change to what you want
;             :::: :::.-- 0-3
      lda   #%0001_1010
      sta   control

;The ACIA "command" register controls the parity, echo mode, transmit and
;receive interrupt enabling, hardware "BRK", and (indirectly) the "RTS"
;and "DTR" lines.  The value below sets the ACIA for no parity check,
;no echo, disables transmit interrupts, and enables receive interrupts
;(RTS and DTR low).
;             .------------------------- parity control,
;             :.------------------------ bits 5-7
;             ::.----------------------- 000 = no parity
;             :::
;             :::.------------------- echo mode, 0 = normal (no echo)
;             ::::
;             :::: .----------- transmit interrupt control, bits 2-3
;             :::: :.---------- 10 = xmit interrupt off, RTS low
;             :::: ::
;             :::: ::.------ receive interrupt control, 0 = enabled
;             :::: :::
;             :::: :::.--- DTR control, 1=DTR low
      lda   #%0000_1001
      sta   command

;Besides initialization, also call the following code whenever the user
;changes parity of echo mode.
;It creates the "xmitoff" and "xmiton" models used by the interrupt
;handler and main-program transmit routine to control the ACIA
;interrupt enabling.  If you don't change the models' parity bits,
;you'll revert to "default" parity on the next NMI.

                        ;initialize with transmit interrupts off since
                        ;buffer will be empty

      sta   xmitoff     ;store as a model for future use
      and   #%1111_0000 ;mask off interrupt bits, keep parity/echo bits
      ora   #%0000_0101 ;and set bits to enable both transmit and
                        ;receive interrupts
      sta   xmiton      ;store also for future use

;The standard NMI routine tests th <RESTORE> key, CIA #2, and checks
;for the presence of an autostart cartridge.

;You can safely bypass the normal routine unless:
;       *  you want to keep the user port active
;       *  you want to use the TOD clock in CIA #2
;       *  you want to detect an autostart cartridge
```

```
;       *  you want to detect the RESTOR key
;
;If you need any of these functions, you can wedge the ACIA
;interrupt handler in ahead of the Kernal routines.  It's probably
;safer to replicate in your own program only the Kernal NMI functions
;that you need.  We'll illustrate bypassing all Kernal tests.

;BE SURE THE "NEWNMI" ROUTINE IS IN PLACE BEFORE EXITING THIS CODE!
;A "stray" NMI that occurs after the vector is changed to NEWNMI's address
;will probably cause a system crash if NEWNMI isn't there.  Also, it would
;be best to initialize the ACIA to a "no interrupts" state until the
;new vector is stored.  Although a power-on reset should disable all
;ACIA interrupts, it pays to be sure.

;If the user turns the modem off and on, an interrupt will probably be
;generated.  At worst, this may leave a stray character in teh receive
;buffer, unless you don't have NEWNMI in place.

NEWVEC:
      sei               ;A stray IRQ shouldn't cause any problems
                        ;while we're changing the NMI vector, but
                        ;why take chances?

;If you want all the normal NMI tests to occur after the ACIA check,
;save the old vector.  If you don't need the regular stuff, you can
;skip the next four lines.  Note that the Kernal NMI routine pushes
;the CPU registers to the stack.  If you call it at the normal address,
;you should pop the registers first (see EXITINT below).

      lda   NMINV       ;get low byte of present vector
      sta   OLDVEC      ;and store it for future use
      lda   NMINV+1     ;do the same
      sta   OLDVEC+1    ;with the high byte

                        ;come here from the SEI if you're not saving
                        ;the old vector
      lda   #<NEWNMI    ;get low byte of new NMI routine
      sta   NMINV       ;store in vector
      lda   #>NEWNMI    ;and do the same with
      sta   NMINV+1     ;the high byte

      cli               ;allow IRQs again

;continue initializing your program

;     :::   ::::::      ;program initialization continues
      jmp   TERMINAL    ;go to the example dumb-terminal subroutine

;Save two bytes to store the old vector only if you need it


;    ---=== New NMI Routine Starts Here ===---

;The code below is a simple interrupt patch to control the ACIA.  When
;the ACIA generates an interrupt, this routine examines the status
;register which contains the following data.

;             .---------------------------- high if ACIA caused interrupt;
;             :                             not used in code below
;             :
;             :.------------------------- reflects state of DCD line
;             ::
;             ::.---------------------- reflects state of DSR line
;             :::
;             :::.------------------ high if xmit-data register is empty
;             ::::
;             :::: .--------------- high if receive-data register full
;             :::: :
;             :::: :.----------- high if overrun error
;             :::: ::
;             :::: ::.------- high if framing error
;             :::: :::
;             :::: :::.--- high if parity error
;     status  xxxx_xxxx

NEWNMI:
;     sei               ;the Kernal routine already does this before jumping
                        ;through the NMINV vector
      pha               ;save A register
      txa
      pha               ;save X register
```

```
      tya
      pha              ;save Y register

;As discussed above, the ACIA can generate an interrupt from one of four
;different sources.  We'll first check to see if the interrupt was
;caused by the receive register being full (bit #3) or the transmit
;register being empty (bit #4) since these two activities should receive
;priority.  A BEQ (Branch if EQual) tests the status register and branches
;if the interrupt was not caused by the data register.

;Before testing for the source of the interrupt, we'll prevent more
;interrupts from the ACIA by disabling them at the chip.  This prevents
;another NMI from interrupting this one.  (SEI won't work because the
;CPU can't disable non-maskable interrupts).

;At lower baud rates (2400 baud and lower) this may not be necessary.  But,
;it's safe and doesn't take much time, either.

;The same technique should be used in parts of your program where timing
;is critical.  Disk access, for example, uses SEI to mask IRQ interrupts.
;You should turn off the ACIA interrupts during disk access also to prevent
;disk errors and system crashes.

;First, we'll load the status register which contains all the interrupt
;and any received-data error information in the 'A' register.

      lda   status

;Now prevent any more NMIs from the ACIA

      ldx   #%0000_0011   ;disable all interrupts, bring RTS inactive, and
                          ;leave DTR active
      stx   command       ;send to ACIA-- code at end of interrupt handler
                          ;will re-enable interrupts

;Store the status-register data only if needed for error checking.
;The next received byte will clear the error flags.

;     sta   errors        ;only if error checking implemented

      and   #%0001_1000   ;mask out all but transmit and
                          ;receive interrupt indicators

;If you don't use a transmit buffer you can use
;
;     and   #%0000_1000
;
;to test for receive interrupts only and skip the receive test shown
;below.

      beq   TEST_DCD_DSR

;if the 'A' register=0, either the interrupt was not caused by the
;ACIA or the ACIA interrupt was caused by a change in the DCD or
;DSR lines, so we'll branch to check those sources.

;If your program ignores DCD and DSR, you can branch to
;the end of the interrupt handler instead:
;
;     beq   NMIEXIT
;

;Test the status register information to see if a received byte is ready
;If you don't use a transmit buffer, skip the next two instructions.

RECEIVE:                  ;process received byte
      and   #%0000_1000   ;mask all but bit #3
      beq   XMITCHAR      ;if not set, no received byte - if you're using
                          ;a transmit buffer, the interrupt must have been
                          ;caused by transmit.  So, branch to handle.
      lda   data          ;get received byte
      ldy   rtail         ;index to buffer
      sta   (rbuff),y     ;and store it
      inc   rtail         ;move index to next slot
      inc   recvcount     ;increment count of bytes in receive buffer
                          ;(if used by your program)

;Skip the "XMIT" routines below if you decide not to use a transmit buffer.
;In that case, the next code executed starts at TEST_DCD_DSR or NMIEXIT.

;After processing a received byte, this sample code tests for bytes
```

```
;in the transmit buffer and sends on if present.  Note that, in this
;sample, receive interrupts take precedence.  You may want to reverse the
;order in your program.

;If the ACIA generated a transmit interrupt and no received byte was
;ready, status bit #4 is already set.  The ACIA is ready to accept
;the byte to be transmitted and we've branched directly to XMITCHAR below.

;If only bit #3 was set on entry to the interrupt handler, the ACIA may have
;been in the process of transmitting the last byte, and there may still be
;characters in the transmit buffer.  We'll check for that now, and send the
;next character if there is one.  Before sending a character to the ACIA to
;be transmitted, we must wait until bit #4 of the status register is set.

XMIT:
      lda   xmitcount     ;if not zero, characters still in buffer
                          ;fall through to process xmit buffer
      beq   TEST_DCD_DSR  ;no characters in buffer-- go to next check
;or
;
;     beq   NMIEXIT
;
;if you don't check DCD or DSR in your program.

XMITBYTE:
      lda   status        ;test bit #4
      and   #%00010000
      beq   TEST_DCD_DSR  ;skip if transmitter still busy

XMITCHAR:                 ;transmit a character
      ldy   thead
      lda   (tbuff),y     ;get character at head of buffer
      sta   data          ;place in ACIA for transmit

                          ;point to next character in buffer
      inc   thead         ;and store new index
      dec   xmitcount     ;subtract one from count of bytes
                          ;in xmit buffer
      lda   xmitcount
      beq   TEST_DCD_DSR
;or
;
;     beq   NMIEXIT
;
;if you don't check DCD or DSR in your program

;If xmitcount decrements to zero, there are no more characters to
;transmit.  The code at NMIEXIT turns ACIA transmit interrupts off.

;If there are more bytes in the buffer, set up the 'A' register with
;the model that turns both transmit and receive interrupts on.  We felt
;that was safer, and not much slower, than EORing bits #3 and #4.  Note
;that the status of the parity/echo bits is preserved in the way "xmiton"
;and "xmitoff" were initialized earlier.

      lda   xmiton        ;model to leave both interrupts enabled

;If you don't use DCD or DSR

      bne   NMICOMMAND    ;branch always to store model in command register

;If your program uses DCD and/or DSR, you'll want to know when the state
;of those lines changes.  You can do that outside the interrupt handler
;by polling the ACIA status register, but if either of the lines changes,
;the chip will generate an NMI anyway.  So, you can let the interrupt
;handler do teh work for you.  The cost is the added time required to
;execute the DCD_DSR code on each NMI.

TEST_DCD_DSR:

;     pha                 ;only if you use a transmit buffer, 'A' holds
                          ;the proper mask to re-enable interrupts on
                          ;the ACIA
;     ::
;     ::                  ;appropriate code here to compare bit #6 (DCD)
;     ::                  ;and/or bit #5 (DSR) with their previous states
;     ::                  ;which you've already stored in memory and take
;     ::                  ;appropriate action
;     ::
;     pla                 ;only if you pushed it at the start of the
                          ;DCD/DSR routine
```

```
;     bne   NMICOMMAND    ;'A' holds the xmiton mask if it's not zero,
                          ;implying that we arrived here from xmit routine
                          ;not used if you're not using a transmit buffer.

;If the test for ACIA interrupt failed on entry to the handler, we branch
;directly to here.  If you don't use additional handlers, the RESTORE key
;(for example) will fall through here and have no effect on your program
;or the machine, except for some wasted cycles.

NMIEXIT:
      lda   xmitoff       ;load model to turn transmit interrupts off

;and this line sets the interrupt status to whatever is in the 'A' register.

NMICOMMAND:
      sta   command

;That's all we need for the ACIA interrupt handler.  Since we've pushed the
;CPU registers to the stack, we need to pop them off.  Note that you must
;do this EVEN IF YOU JUMP TO THE KERNAL HANDLER NEXT, since it will push
;them again immediately.  You can skip this step only if you're proceeding
;to a custom handler.

EXITINT:                  ;restore things and exit
      pla                 ;restore 'Y' register
      tay
      pla                 ;restore 'X' register
      tax
      pla                 ;restore 'A' register

;If you want to continue processing the interrupt with the Kernal routines,

      jmp   (OLDVEC)      ;continue processing interrupt with Kernal handler

;Or, if you add your own interrupt routine,

;     jmp   YOURCODE      ;continue with your own handler

;If you use your own routine, or if you don't add anything, BE SURE to do
;this last (C64 only):

;     rti                 ;return from interrupt instruction

;to restore the flags register the CPU pushes to the stack before jumping
;to the Kernal code.  It also returns you to the interrupted part of
;your program

;-------------------------------------------------------------------------
;Sample routine to store a character in the buffer to be transmitted
;by the ACIA.

;(c) 1990 by Noel Nyman, Kent Sullivan, Brian Minugh,
;Geoduck Developmental Systems, and Dr. Evil Labs.

;Assumes the character is in the 'A' register on entry.  Destroys 'Y'--
;push to stack if you need to preserve it.

SENDBYTE:                 ;adds a byte to the xmit buffer and sets
                          ;the ACIA to enable transmit interrupts (the
                          ;interrupt handler will disable them again
                          ;when the buffer is empty)

      ldy   xmitcount     ;count of bytes in transmit buffer
      cpy   #255          ;max buffer size
      beq   NOTHING       ;buffer is full, don't add byte

      ldy   ttail         ;pointer to end of buffer
      sta   (tbuff),y     ;store byte in 'A' at end of buffer
      inc   ttail         ;point to next slot in buffer
      inc   xmitcount     ;and add one to count of bytes in buffer

      lda   xmiton        ;get model for turning on transmit interrupts
      sta   command       ;tell ACIA to do it

      rts                 ;return to your program

NOTHING:
      lda   #$00          ;or whatever flag your program uses to tell that the
                          ;byte was not transmitted
      rts                 ;and return
```

```
;Alternative routine to transmit a character from main program when not using
;a transmit buffer.
;
;(c) 1990 by Noel Nyman, Kent Sullivan, Brian Minugh,
;Geoduck Developmental Systems, and Dr. Evil Labs.
;
;Assumes the character to be transmitted is in the 'A' register on entry.
;Destroys 'Y'; push to stack if you need to preserve it.
;
;SENDBYTE:
;     tay                 ;remember byte to be transmitted
;
;TESTACIA:
;     lda   status        ;bit #4 of the status register is set if
;                         ;the ACIA is ready to transmit another byte,
;                         ;even if transmit interrupts are disabled.
;     and   #%0001_0000
;     beq   TESTACIA      ;wait for bit #4 to be set
;     sty   data          ;give byte to ACIA
;     rts

;Sample routine to fetch a character that has been received, from the
;receive buffer.

;by Craig Bruce, 1995, adapted from above

;Will return the character in the 'A' register and the carry flag cleared if
;a character was available.  If no character was available, will return with
;the carry flag set.  Destroys the 'Y' register.

RECVBYTE:                ;fetches a byte from the receive buffer.
                         ;there is no need to fiddle with any interrupts

      lda   recvcount    ;count of bytes in receive buffer
      beq   RECVEMPTY    ;buffer is empty, indicate to caller

      ldy   rhead        ;pointer to start of buffer
      lda   (rbuff),y    ;fetch byte out of buffer into 'A' register
      inc   rhead        ;point to next slot in buffer
      dec   recvcount    ;and add one to count of bytes in buffer

      clc                ;indicate that we have a character
      rts                ;return to your program

RECVEMPTY:
      sec                ;or whatever flag your program uses to tell that the
                         ;receive buffer was empty
      rts                ;and return

;---------------------------------------------------------------------------
;Dumb -- very dumb -- terminal emulator.  Simply polls the receive buffer and
;the keyboard and puts received data to the screen and typed data to the send
;buffer (thus, it assumes a full-duplex, echoing link).  There is no
;PETSCII->ASCII conversion, no cursor, nor any other fancy features.  Press
;STOP to exit.
;
;by Craig Bruce, 1995.

TERMINAL:
      jsr   RECVBYTE     ;see if there is a received byte in the recv buffer
      bcs   TERMTRYSEND  ;if not, continue
      jsr   $FFD2        ;if received byte, print it to the screen (CHROUT)
TERMTRYSEND:
      jsr   $FFE4        ;try to get a character from the keyboard (GETIN)
      cmp   #$00         ;was there a keystroke available?
      beq   TERMINAL     ;no--go back to top of polling loop
      cmp   #$03         ;check for STOP key
      beq   TERMEXIT     ;  exit if pressed
      jsr   SENDBYTE     ;have char--put it into the transmit buffer and then
      jmp   TERMINAL     ;  go back to top of polling loop
TERMEXIT:
      lda   #%0000_0010  ;disable transmitter and receiver and all interrupts
      sta   command
      sei
      lda   OLDVEC       ;restore the NMI vector to its original value
      sta   NMINV
      lda   OLDVEC+1
      sta   NMINV+1
      cli
      rts                ;exit
```

```
TRANSMIT_BUFFER = *+0
RECEIVE_BUFFER  = *+256
%%%---8<---cut-here---8<---%%%
```

APPENDIX: 6551 ACIA HARDWARE SPECS (DATA SHEET)

C= Commodore Semiconductor Group
a division of Commodore Business Machines, Inc.
950 Rittenhouse Road, Norstown, PA 19400 * 215/666-7950 * TWX 510-660-4168
(July 1987)

6551 ASYNCHRONOUS COMMUNICATION INTERFACE ADAPTER

CONCEPT:

The 6551 is an Asynchronous Communication Adapter (ACIA) intended to provide for interfacing the 6500/6800 microprocessor families to serial communication data sets and modems. A unique feature is the inclusion of an on-chip programmable baud-rate generator, with a crystal being the only external component required.

FEATURES:

* On-chip baud-rate generator: 15 programmable baud rates derived from a standard standard 1.8432 MHz external crystal (50 to 19,200 baud) [these rates are doubled in the SwiftLink].

* Programmable interrupt and status register to simplify software design.

* Single +5 volt power supply.

* Serial echo mode.

* False start bit detection.

* 8-bit bi-directional data bus for direct communication with the microprocessor.

* External 16x clock input for non-standard baud rates (up to 125 Kbaud).

* Programmable: word lengths; number of stop bits; and parity-bit generation and detection.

* Data set and modem control signals provided.

* Parity: (odd, even, none, mark, space).

* Full-duplex or half-duplex operation.

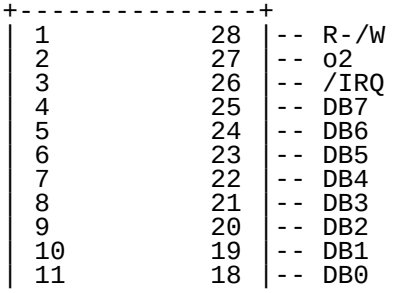
* 5,6,7 and 8-bit transmission.

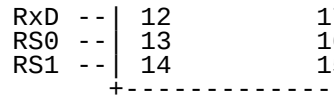
* 1-MHz, 2-MHz, and 3-MHz operation.

ORDER NUMBER

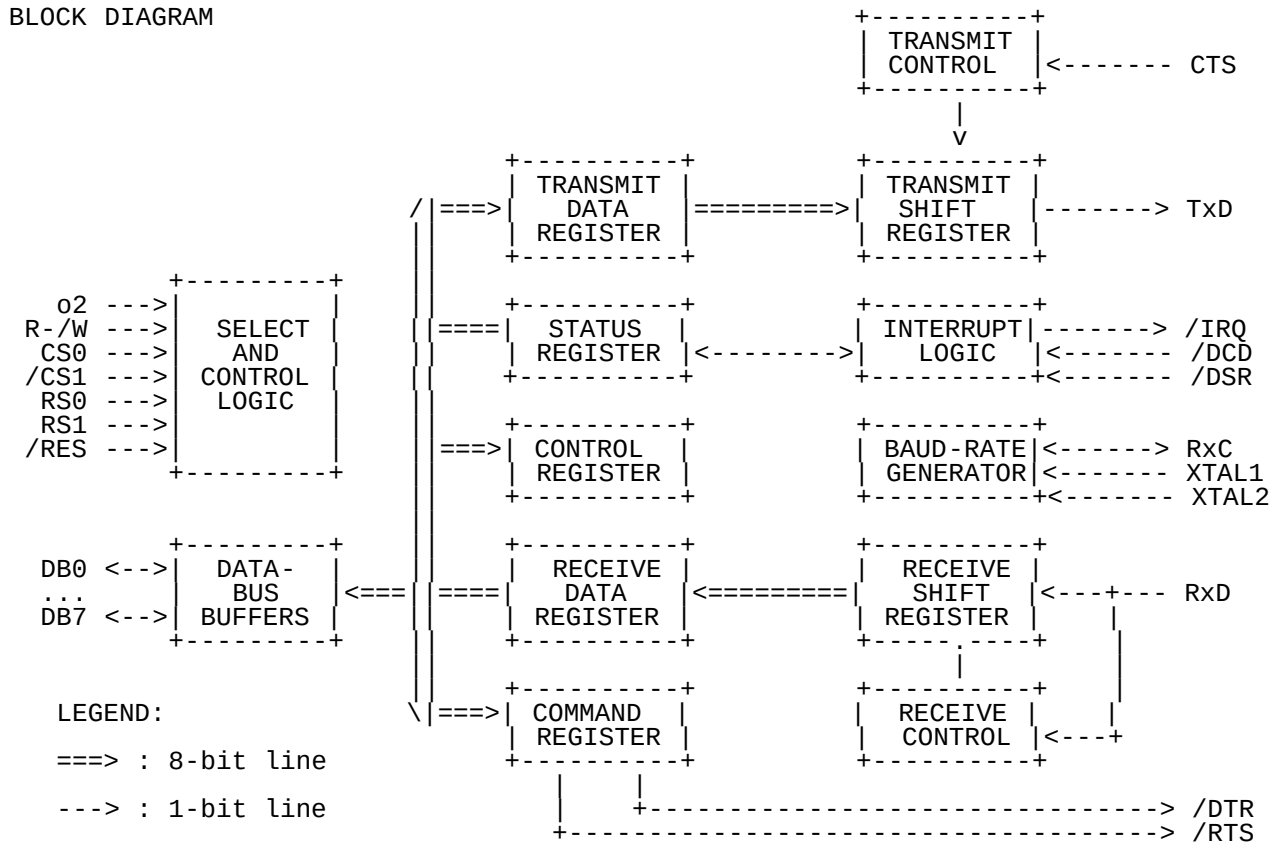
```
MXS 6551 ___
 -        |
 |        +---- Frequency range
 |                   Plain = 1 MHz
 |                       A = 2 MHz
 |                       B = 3 MHz
 |
 +------------ Package Designator
                     C = Ceramic
                     P = Plastic
```

6551 PIN CONFIGURATION





BLOCK DIAGRAM



MAXIMUM RATINGS

<not included here>

ELECTRICAL CHARACTERISTICS

<not included here>

POWER DISSIPATION vs TEMPERATURE

<not included here>

TIMING CHARACTERISTICS

<not included here>

INTERFACE SIGNAL DESCRIPTION

/RES (Reset)

During system initialization a low on the /RES input will cause internal registers to be cleared.

o2 (Input Clock)

The input clock is the system o2 clock and is used to trigger all data transfers between the system microprocessor and the 6551.

R-/W (Read/Write)

The R-/W is generated by the microprocessor and is used to control the direction of data transfers. A high on the R-/W pin allows the processor to read the data supplied by the 6551. A low on the R-/W pin allows a write to the 6551.

/IRQ (Interrupt Request)

The /IRQ pin is an interrupt signal from the interrupt-control logic. It is an open drain output, permitting several devices to be connected to the common /IRQ microprocessor input. Normally a high level, /IRQ goes low when an

interrupt occurs.

DB0--DB7 (Data Bus)

The DB0--DB7 pins are the eight data lines used for transfer of data between the processor and the 6551. These lines are bi-directional and are normally high-impedance except during Read cycles when selected.

CS0, /CS1 (Chip Selects)

The two chip-select inputs are normally connected to the processor-address lines either directly or through decoders. The 6551 is selected when CS0 is high and /CS1 is low.

RS0, RS1 (Register Selects)

The two register-select lines are normally connected to the processor-address lines to allow the processor to select the various 6551 internal registers. The following table indicates the internal register-select coding:

RS1	RS0	WRITE	READ	SL-Addr
0	0	Transmit Data Register	Receive Data Register	$DE00
0	1	Programmed Reset*	Status Register	$DE01
1	0	Command Register	Command Register	$DE02
1	1	Control Register	Control Register	$DE03

* for programmed reset, data is "don't care".

The table shows that only the Command and Control registers are read/write. The Programmed Reset operation does not cause any data transfer, but is used to clear the 6551 registers. The Programmed Reset is slightly different from the Hardware Reset (/RES) and these differences are described in the individual register definitions.

ACIA/MODEM INTERFACE SIGNAL DESCRIPTION

XTAL1, XTAL2 (Crystal Pins)

These pins are normally directly connected to the external crystal (1.8432 MHz) used to derive the various baud rates. Alternatively, an externally generated clock may be used to drive the XTAL1 pin, in which case the XTAL2 pin must float. XTAL1 is the input pin for the transmit clock.

TxD (Transmit Data)

The TxD output line is used to transfer serial NRZ (non-return-to-zero) data to the modem. The LSB (least-significant bit) of the Transmit Data Register is the first data bit transmitted and the reate of data transmission is determined by the baud rate selected.

RxD (Receive Data)

The RxD input line is used to transfer serial NRZ data into the ACIA from the modem, LSB first. The receiver data rate is either the programmed baud rate or the rate of an externally generated receiver clock. This selection is made by programming the Control Register.

RxC (Receive Clock)

The RxC is a bi-directional pin which serves as either the receiver 16x clock input or the receiver 16x clock output. The latter mode results if the internal baud rate generator is selected for receiver data clocking.

/RTS (Request to Send)

The /RTS output pin is used to control the modem from the processor. The state of the /RTS pin is determined by the contents of the Command Register.

/CTS (Clear to Send)

The /CTS input pin is used to control the transmitter operation. The enable state is with /CTS low. The transmitter is automatically disabled if /CTS is high.

/DTR (Data Terminal Ready)

The output pin is used to indicate the status of the 6551 to the modem. A low of /DTR indicates the 6551 is enabled and a high indicates it is disabled. The processor controls this pin via bit 0 of the Command Register.

/DSR (Data Set Ready)

The /DSR input pin is used to indicate to the 6551 the status of the modem. A low indicates the "ready" state and a high, "not-ready". /DSR is a high-impedance input and must not be a no-connect. If unused, it should be driven high or low, but not switched.

Note: If Command Register Bit #0 = 1 and a change of state on /DSR occurs, /IRQ will be set and Status Register Bit #[5] will reflect the new level. The state of /DSR does not affect Transmitter operation [but must be low for the Receiver to operate]. [This statement reflects the SwiftLink implementation].

/DCD (Data Carrier Detect)

The /DCD input pin is used to indicate to the 6551 the status of the carrier-detect output of the modem. A low indicates that the modem carrier signal is present and a high, that it is not. /DCD, like /DSR, is a high-impedance input and must not be a no-connect.

Note: If Command Register Bit #0 = 1 and a change of state on /DSR occurs, /IRQ will be set and Status Register Bit #[6] will reflect the new level. The state of /DCD does not affect either Transmitter or Receiver operation.

INTERNAL ORGANIZATION

<not included here>

TRANSMIT AND RECEIVE DATA REGISTERS (SL-Addr: $DE00 / 56832)

These registers are used as temporary data storage for the 6551 Transmit and Receive circuits. The Transmit Data Register is characterized as follows:

* Bit 0 is the leading bit to be transmitted.

* Unused data bits are the high-order bits and are "don't care" for transmission.

The Receive Data Register is characterized in a similar fashion:

* Bit 0 is the leading bit received.

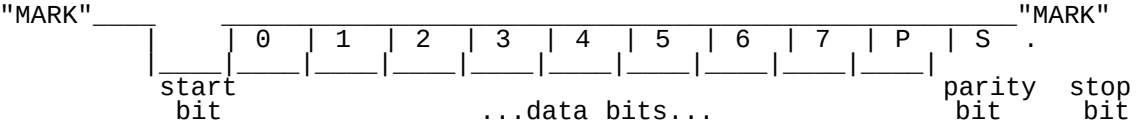
* Unused data bits are the high-order bits and are "0" for the receiver.

* Parity bits are not contained in the Receive Data Register, but are stripped off after being used for external parity checking. Parity and all unused high-order bits are "0".

Transmit / Receive Data Register

7	6	5	4	3	2	1	0
data							

The following figure illustrates a single transmitted or received data word, for the example of 8 data bits, parity, and 1 stop bit:



STATUS REGISTER (SL-Addr: $DE01 / 56833)

The Status Register is used to indicate to the processor the status of various 6551 functions and is outlined here:

Command Register

7	6	5	4	3	2	1	0
irq	dcd	dsr	txr	rxr	ovr	fe	pe

7	/IRQ*** : cleared by reading status register
0	No interrupt
1	Interrupt

```
| 6 |   /DCD : non-resetable, indicates /DCD status
+---+   ----------------------------------------
  0     /DCD low
  1     /DCD high

+---+
| 5 |   /DSR : non-resetable, indicates /DSR status
+---+   ----------------------------------------
  0     /DSR low
  1     /DSR high

+---+
| 4 |   Transmit Data Register Empty: Cleared by write to Tx Data reg
+---+   ------------------------------------------------------------
  0     Not empty
  1     Empty

+---+
| 3 |   Receive Data Register Full: Cleared by read from Rx Data reg
+---+   ------------------------------------------------------------
  0     Not full
  1     Full

+---+
| 2 |   Overrun*: Self-clearing**
+---+   --------------------------
  0     No error
  1     Error

+---+
| 1 |   Framing Error*: Self-clearing**
+---+   --------------------------------
  0     No error
  1     Error

+---+
| 0 |   Parity Error*: Self-clearing**
+---+   -------------------------------
  0     No error
  1     Error

Notes:   * No interrupt generated for these conditions
         ** Cleared automatically after a read of RDR and the next error-
              free receipt of data
        *** Reading status reg. will clear the /IRQ bit except when
              transmit intr. enabled

  7   6   5   4   3   2   1   0
+---+---+---+---+---+---+---+---+
| 0 | x | x | 1 | 0 | 0 | 0 | 0 |  After Hardware reset
+---+---+---+---+---+---+---+---+
| x | x | x | x | x | 0 | x | x |  After Software reset
+---+---+---+---+---+---+---+---+
```

COMMAND REGISTER (SL-Addr: $DE02 / 56834)

The Command Register is used to control specific Transmit/Receive functions and is shown here:

```
                  Command Register
+-----+-----+-----+-----+-----+-----+-----+-----+
|  7  |  6  |  5  |  4  |  3  |  2  |  1  |  0  |
+-----+-----+-----+-----+-----+-----+-----+-----+
|     parity      | echo|  tx ctrl  | rxi | dtr |

+---+---+---+
| 7 | 6 | 5 |   PARITY CHECK CONTROLS
+---+---+---+   ----------------------
  x   x   0     parity disabled--no parity bit generated or received
  0   0   1     odd parity receiver and transmitter
  0   1   1     even parity receiver and transmitter
  1   0   1     mark parity transmitted, parity check disabled
  1   1   1     space parity transmitted, parity check disabled

+---+
| 4 |   NORMAL/ECHO MODE FOR RECEIVER
+---+   ------------------------------
  0     Normal
  1     Echo (bits 2 and 3 must be "0")
```

```
+---+---+
| 3 | 2 |   Tx INTERRUPT    RTS LEVEL    TRANSMITTER
+---+---+   ------------    ---------    ------------
  0   0       Disabled         High           Off
  0   1       Enabled          Low            On
  1   0       Disabled         Low            On
  1   1       Disabled         Low       Transmit BRK

+---+
| 1 |   RECEIVE INTERRUPT ENABLE
+---+   -------------------------
  0     /IRQ interrupt Enabled from bit 3 of Status Register
  1     /IRQ interrupt Disabled

+---+
| 0 |   DATA TERMINAL READY
+---+   --------------------
  0     Disable Receiver and all interrupts (/DTR high)
  1     Enable Receiver and all interrupts  (/DTR low)

  7   6   5   4   3   2   1   0
+---+---+---+---+---+---+---+---+
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  After Hardware reset
+---+---+---+---+---+---+---+---+
| x | x | x | 0 | 0 | 0 | 0 | 0 |  After Software reset
+---+---+---+---+---+---+---+---+
```

CONTROL REGISTER (SL-Addr: $DE03 / 56835 / cpm: 0001xxxx)

The Control Register is used to select the desired mode for the 6551. The word length, number of stop bits, and clock controls are all determined by the Control Register, which is shown here:

```
                  Control Register
+-----+-----+-----+-----+-----+-----+-----+-----+
|  7  |  6  |  5  |  4  |  3  |  2  |  1  |  0  |
+-----+-----+-----+-----+-----+-----+-----+-----+
|stops|  word len | src |       baud rate       |

+---+
| 7 |   STOP BITS
+---+   ----------
  0     1 stop bit
  1     2 stop bits
  1     1 stop bit if word length== 8 bits and parity
            this allows for 9-bit transmission (8 data bits plus parity)
  1     1.5 stop bits if word length== 5 bits and no parity

+---+---+
| 6 | 5 |   WORD LENGTH
+---+---+   ------------
  0   0     8 bits
  0   1     7 bits
  1   0     6 bits
  1   1     5 bits

+---+
| 4 |   RECEIVER CLOCK SOURCE
+---+   ----------
  0     external receiver clock
  1     baud rate generator

+---+---+---+---+
| 3 | 2 | 1 | 0 |   BAUD RATE GENERATOR
+---+---+---+---+   --------------------
  0   0   0   0     16x external clock
  0   0   0   1     100 baud
  0   0   1   0     150 baud
  0   0   1   1     219.84 baud
  0   1   0   0     269.16 baud
  0   1   0   1     300 baud
  0   1   1   0     600 baud
  0   1   1   1     1200 baud
  1   0   0   0     2400 baud
  1   0   0   1     3600 baud
  1   0   1   0     4800 baud
  1   0   1   1     7200 baud
  1   1   0   0     9600 baud
  1   1   0   1     14400 baud
  1   1   1   0     19200 baud
```

```
   1   1   1   1     38400 baud

   7   6   5   4   3   2   1   0
 +---+---+---+---+---+---+---+---+
 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  After Hardware reset
 +---+---+---+---+---+---+---+---+
 | x | x | x | x | x | x | x | x |  After Software reset
 +---+---+---+---+---+---+---+---+
```

==========

Design and Implementation of a Simple/Efficient Upload/Download Protocol
by Craig Bruce <csbruce@ccnga.uwaterloo.ca>

1. INTRODUCTION

If you use your Commodore for telecommunications, then you are basically interested in two things: using your C= to emulate a terminal for interactive stuff, and using modem-file-transfer protocols to upload and download files from and to your Commodore.

This document describes a custom upload/download protocol that was designed for use with the ACE-128/64 system and is freely available to anyone who wants it (well, when I finish with the Release #14 of ACE). While this protocol non-standard, it blows the doors off of all other protocols available for Commodore computers, even though it uses a simple "stop-and-wait" acknowledgement scheme. There are two reasons for its speed: the fast device drivers available with ACE, and its large packet size, up to about 18K (although this could be significantly larger is ACE's memory usage were reorganized).

The name of the protocol is "Craig's File eXchange Protocol", or just "FX" for short. It is "file exchange" rather than "upload" or "download" because you will use the same activation of the program to both upload and download all of the files you name.

2. USAGE

The current implementation of FX consists of a "client" program for you to run on your Commodore computer and a "server" program that you run on your Unix host. There is currently no server program for any other platform, but the necessary changes to the C-language program wouldn't be too hard. The client program is written in 6502 assembler, of course (for the ACE-assembler to be specific).

FX is an external program from the terminal program, so (for now) to activate FX, you have to exit from the terminal program and enter the FX command line, exchange the files, and then re-enter the terminal program from the command line.

When you run FX, you will activate the Server program first on your Unix host and then exit the terminal program and run the Client program on your Commodore. You run the command "fx" on both the client and server machines, which may be a little confusing (but I think you'll get used to it), and name the files that you want to have transferred as arguments to the command on the machine that you want to transfer the files FROM. The usage of the "fx" command is as follows:

```
fx [-dlvV7] [-m maximums] [-f argfile] [[-b] binfile ...] [-t textfile ...]

-d = debug mode
-l = write to log file ("fx.log")
-v = verbose log/debug mode
-V = extremely verbose log/debug mode
-7 = use seven-bit encoding
-m = set maximum packet sizes; maximums = ulbin/ultxt/dlbin/dltxt (bytes)
-f = take arguments one-per-line from given argfile
-b = binary files prefix
-t = text files prefix
-help = help
```

well, for the server, anyway. The client program doesn't have the more exotic options. The "-d", "-l", "-v", and "-V" options are available only on the Server program, and are for debugging purposes only.

The "-7" option tells the protocol to use only 7-bit data. I.e., it tells it to not use the 8th bit position in the data is transmitted. This is useful if you are forced into the humiliation of only being able to use a 7-bit channel to your Unix host. You need only need to give this option on either the client or the host command line and the other side will be informed. It may be useful to create an alias for this command with all of your options set to what you want them to be.

The protocol has the capacity to use different packet sizes for four types of file-transfer situations: uploading binary data, uploading text, downloading binary data, and downloading text. These are useful distinctions, since your host may or may not be able to handle the larger packet sizes without losing bytes (your Commodore, of course, can handle the larger packet sizes with no problems).

In determining which packet size to use for a file transfer (where the type of transfer is known), the protocol finds that largest packet size that both the client and the server can handle and then take the minimum of these two values. The defaults for the client are all the same: the maximum amount of program-area memory that it can use, about 18K. For the server program, I have programmed in default maximum uploading packet sizes of 1K and maximum downloading packet sizes of 64K-1. You can change these defaults in the C program easily by changing some "#define"s.

The "-m" option allows you to manually set the default packet sizes for a transfer. The argument following the "-m" flag should have four numbers with slashes between them, which give the maximum ulbin/ultxt/dlbin/dltxt packet sizes, respectively. Note that the packet sizes only include the size of the user data encoded into packets and not the control or quoting information (below).

The "-f" option on the server allows you to read arguments from a file rather than the command line. This is useful if want to generate and edit the list of files to download before you run the FX command. It's also useful if you don't want other users to see the names of the files that you are downloading. The name of the file comes in the first argument following the "-f" flag and the arguments are put into this file one-per-line. You can put in "-" options in addition to filenames if you wish (like "-t" and "-b"). This option is not supported on the client program.

Finally come the "-b", "-t", and filename arguments. The "-b" argument tells FX that all of the following filenames (until the next "-t" option) are binary files and the "-t" argument says that the following filenames are all of text files. You can use as many "-b" and "-t" arguments as you want. If you don't use any, then all of the files you name will be assumed to be binary files.

For each filename you give on a command line, that file will be transferred from that machine to the other machine. On both Unix and ACE, you can use wildcards in your filenames, of course, to transfer groups of files.

The client program controls the file exchange, and it uploads all of its files first and then asks the server if the server has any files to be downloaded. When the exchange is completed, both the client and server FX programs will exit and you will find yourself back on the command lines in both environments. Re-enter the terminal program to continue with your online session. If something goes very wrong during a transfer or if you decide that you don't really want to transfer any files after activating the server program, you can type three Ctrl-X's to abort the server. This is the same as for the X-modem protocol.

3. DESIGN DECISIONS

There are a number of design decisions to be made about our protocol. But first, we want to recognize and appreciate that since we have a license to design a completely new protocol, we are not bound, shackled, gagged, and tortured by the "hysterical raisins" and bad design decisions of existing compromised and bloated standard protocols... such as Z-modem.

We want the protocol to understand whether a file is text or binary data and to translate them appropriately during downloading. We want the protocol to be aware of filenames, dates, permissions, and we do not want our file contents to get mangled like they do with X-modem (it pads them with Ctrl-Z's, since it was designed for CP/M), and we want it to translate to/from PETSCII if the file is text. We will require that the user tell us whether the file is binary or text (although we may be able to statistically determine this from snooping through the file), and we will use a "canonical form" for encoding the text data during transfer. A convenient canonical form to use is Unix-ASCII (ASCII-LF).

We want our protocol to be simultaneously simple and fast. To make it simple, we will use a stop-and-wait acknowledgement scheme. This means that after each packet is uploaded or downloaded, the transfer will pause and wait for the receiving host to acknowledge that the packet has been transferred correctly, and only then will the protocol continue to transfer more data.

In fact, this scheme fits well with the Commodore hardware, since it is not possible to send or receive serial data while doing disk I/O (in the general case), so we would have to stop listening anyway; the protocol makes it so

that there will be no bytes that we end up ignoring while doing I/O.

To make the protocol be fast even though we are using a stop-and-wait acknowledgement scheme, we will use the largest data-packet sizes that we possibly can. In the (current) ACE environment, this means about 18K. This will maximize the amount of time of transferring data over the modem between pauses to do I/O. If the I/O is to the ACE ramdisk, then the length of this pause will be very short and we will achieve a very high link utilization. (The ACE ramdisk can process an 18K read/write request in about 20 milliseconds on a Fast-mode C128 using an REU --- RAMDOS in the same environment would require about 9 _seconds_ (450x slower)).

To allow for future use with other platforms, we will make the protocol define the packet sizes using 32-bit fields. There isn't much data overhead, and this allows us to change implementations to be able to transfer entire files in one large packet. Also, the size of an individual packet should be flexible: be from one to N bytes. This eliminates the X-modem padding problem and the Y-modem crufty hack of using the small packet size when less than 1K of user data remains to be transferred.

We also want our data to be well protected against corruption. Detecting transmission errors efficiently on Commodore computers is already a well solved problem: we will use a table-driven CRC-32 algorithm, the same one that ZMODEM, PKZIP, and CRC32 use. To hide the computation costs of the CRC even more (the cost is very low anyway), we will compute it WHILE sending or receiving packets. Oh, actually, I guess that I forgot to mention an a-priori design decision: we will be using a packet-oriented approach for transferring data (described below); packetization offers so many advantages that this decision is really a no-brainer.

Also, to make the process interaction as straightforward as possible, we want to use the Client/Server programming paradigm. This paradigm combines well with the stop-and-wait acknowledgement scheme to produce a Remote Procedure Call (RPC) type of interaction between the machines. For those not familiar with this Interprocess Communication (IPC) scheme, you can read a couple issues of C= hacking ago where I talked about it for use with a multitasking operation system. RPC is a very useful, powerful, simple, and widely applicable IPC scheme.

To recover from packet corruption, we will be using a timeout+retransmission scheme, and to be consistent with the RPC scheme, the client will do all timeouts and retransmissions. This means that after sending a request RPC packet out, if we don't receive the reply within a certain period of time, we will timeout and send the request again. Or, to be more precise, since we will be working with large packet sizes, we will timeout if we don't receive any bytes from the server for a certain period of time, say 5 seconds, while we are expecting more bytes from him.

The way that corrupted packets are dealt with is very simple: they are ignored. The server could possibly send back a negative acknowledgement, but we won't try that for now.

In order to make retransmissions work out correctly, we will be using sequence numbers and internal-state variables inside of the server to insure that requests aren't carried out more than once. We need these mechanisms because when an RPC fails, we won't know if we got no response because the original request was lost and the operations wasn't carried out, or whether the request was received and carried out but the reply message was lost.

For example, if we request that packet #123 be downloaded and the server carries out that request but the reply message is lost, then the client will time out and retransmit the request. The server remembers the last request number that the client sent it (123 here), so if the client asks for packet #123 again, the server will simply retransmit the reply that it gave last time. If, on the other hand, the client were to request packet #124 (or simply "not 123"), then the server reads the next chunk of data from the file and sends it as the reply. Our protocol will use an 8-bit sequence number even though it only needs a 1-bit sequence number (since eight bits will allow for the future expansion of having multiple requests being processed concurrently: asynchronous RPC).

We also want to be able to both upload and download as conveniently as possible. To me, this means doing both operations by calling only one command (as described in the previous section). This arrangement also allows for the future expansion of uploading and downloading files _simultaneously_ (the protocol as designed places no restrictions on this possibility).

We also want to make use of an eight-bit clean link between the Unix host and your Commodore, but this may not always be possible. Sometimes you may have only a 7-bit connection, and even if you do have an 8-bit connection, there may still be some software-flow-control problems with intermediate devices

between your Commodore and your Unix host. So, we want our protocol to not make use of the X-on and X-off characters, and to use only 7-bit characters if it cannot use eight. The way to achieve this is called "escaping", "quoting" or "byte stuffing", and will be discussed in the next section. It turns out that supporting 7-bit characters is pretty simple and the mechanism is required by other aspects of the packetization.

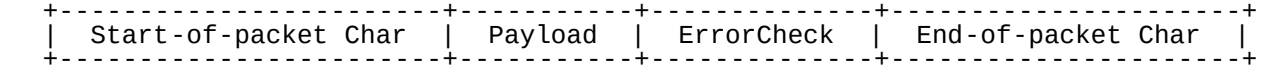
There, that should take care of most of the major design decisions.

4. PACKETIZATION

Packetization refers to the process of taking a stream of data and breaking it up into discrete chunks of data. Each packet is easily identified and is processed as a single unit. There are many general advantages to using packets. If there is a transmission error, then only a single packet is corrupted, and the recovery will be easier since the packet is well identified, and only it needs to be recovered. Packetization also means that a link can be shared between multiple (logical) communication streams fairly and efficiently, and means that a single communication stream can utilize multiple physical links where facilities exist.

Packets also integrate well with many IPC schemes, including Remote Procedure Calls. In fact, you end up emulating a packet-oriented scheme even if you are using RPC over a stream-oriented transport system. Packets also take into account the limited buffering capacity of both end systems and intermediate systems, and allow for the convenient implementation of flow control (even if said flow control consists of simply dropping packets on the floor). Packets are very useful things indeed! And just think that back in the early 1970s packets were dismissed as being infeasible and unusable.

Each packet used in the FX system has four parts to it: the start character, the user data (payload), the error-check characters, and the end character. Graphically, a packet has the following format:



The payload can be arbitrarily long, up to whatever limit the two computers involved in the transfer can handle.

The error check is a 32-bit (4-byte) Cyclic-Redundancy-Check value that occupies the last four bytes before the End-of-packet character. The implementation, which is based on a table-lookup method, is so efficient that it is as fast as a simple add-up checksum, except much more reliable. Using this error check, there will be approximately a one-in-4,000,000,000 chance that a packet with an error in it will be accepted has being error-free. These are pretty good odds for our purposes. The CRC is calculated exclusively on the raw payload data.

The following special characters used by packets are defined:

NAME	HEX	DEC	Control	Meaning
CHR_START	0x01	1	Ctrl-A	Packet-start indicator
CHR_END	0x19	25	Ctrl-Y	Packet-end indicator
CHR_ESC	0x05	5	Ctrl-E	Escape character for next code
CHR_ABORT	0x18	24	Ctrl-X	Abort transfer if repeated three times
CHR_XON	0x11	17	Ctrl-Q	Software flow-start: avoided
CHR_XOFF	0x13	19	Ctrl-S	Software flow-stop: avoided
CHR_QUOTE8	0x14	20	Ctrl-T	Quote-8 the next 7-bit sequence

CHR_START is used to signify the start of a new packet. This character is not allowed to be used anywhere else for any other purpose.

CHR_END is used to signify the end of the current packet, and cannot be used anywhere else. The reason for using special characters to mark the beginning and the ending of a packet is to allow for easy error recovery after a communication failure. All you do is search for the next CHR_START character after you toss away a garbled packet and you're back in business. I am unaware of any reasonable alternatives to framing packets with a CHR_START character. Using a CHR_END special character is a convenience.

CHR_ESC is used to "escape" the next character. Since there are special character codes that cannot be used in any other way than their intended function (including CHR_START and CHR_ESC itself), this character is needed. The character following the CHR_ESC character must be between "@" and "_" (0x40 and 0x5f) in the ASCII chart, or be the character "?" (0x3f). The character following the CHR_ESC is then "and"ed with the value 0x1f to mask off the "letter" bits and turn it into a control character in the range of 0x00 to 0x1f (the same range as the special control characters) and the

"escape sequence" is treated as a single character of user data. If the character following the CHR_ESC is a "?", then a code of 0x7f is interpreted instead. Using a character following the escape that is different from the character being represented allows for greater resiliance of the protocol in the presence of bits being garbled or bytes being dropped. All special characters in a packet except for the starting and ending characters are escaped as described above.

CHR_ABORT can be typed by the user into a terminal program at any time to shut down the server.

CHR_XON and CHR_XOFF can cause problems with intermediate devices on some systems, so the FX protocol does not use these character codes at all; it purposely avoids them and uses escape sequences (CHR_ESC) for them instead.

CHR_QUOTE8 is used to re-generate 8-bit data over a 7-bit link. Kermit uses this same technique. When this character is encountered in the receive stream, the next character is extracted and is "or"ed with a value of 0x80 to give it a "1" in the high-bit position. The CHR_QUOTE8 character can also be followed by a CHR_ESC code, which is interpreted as above and then "or"ed with the 0x80 value.

One of the disadvantages of using this scheme is that each byte in the range of 0x80 and 0xff takes at least two bytes to transmit and some of them three. If fact, for many binary files it may be faster to uuencode the file and transfer the resulting text, since uucode has a static encoding overhead of 33% whereas this quoting scheme has an expected overhead of 50% (plus the CHR_ESC overhead). Of course, this feature is intended to be used as a last resort if you cannot get an 8-bit connection.

So there you have it. Every message sent between the client and the server is encapsulated in a packet as specified above. Packetization allows for convenient error detection and recovery and works well with our interprocess communication scheme.

One implementation note about the packetization has to do with buffering. On the Unix host, it is advantageous to encode a packet into a memory buffer and then send out that buffer in a single "write" operation. This less operating-system overhead (which may or may not be significant) but more importantly, it means that the packet will be sent between intermediate communication devices as efficiently as possible. On my local Unix system, I connect to a terminal server and to my Unix host through that. Performing single-byte writes on the Unix host means that the bytes are sent in individual Ethernet packets between the Unix host and the terminal server, and encounter more overhead and communication delays. When I changed the program to send the FX packet in a single operation, a significant performance gain was realized.

For receiving data on the Unix host, there isn't much you can do other than reading one byte at a time, since the receiver doesn't know when a packet is going to end. However, the same problem is not encountered here that was encountered with sending data because data that is received by the Unix host but not "read" by the user program are buffered and collected, smoothing out the system overhead, which is insignificant compared to the modem speed. The Unix program used the "stdin" and "stdout" file streams for receiving and transmitting data, and sets the tty driver to turn off all line-editing features to get at the raw bytes.

On the Commodore end, it is advantageous to read data from the modem driver in chunks, since the system overhead is significant compared to the modem speed. These are small computers that we are driving to the max, you know. Data is read from the modem in chunks of up to 255 bytes (whatever is available at the time) and processed a byte at a time from the read buffer. The CRC is calculated during processing, to avoid doing this on the critical path. The CRC calculation is performed as an operation by itself since the overhead is very small on fast processors. The character-set translation for text files will be performed on the critical path (on the Commodore) since it is more convenient to do it at a higher layer in the IPC scheme. The packet- handling software is logically at a distinct layer that doesn't have to worry about higher layers. The next layer up is logically the RPC layer and then the file-transfer layer.

5. CLIENT/SERVER OPERATION

As discussed previously, the client/server interaction is based on a Remote Procedure Call paradigm. Thus, for each operation, the client sends a request packet (message) to the server, and the server performs the requested operation and sends back a reply (acknowledgement) message to the client.

There are eight request/ack interactions that are defined for the protocol: two for connection management, three for uploading files, and three for downloading files. The client is in charge of the file-exchange session

and of the error handling.

4.1. CONNECTION MANAGEMENT

When the client starts up, the first thing that it does is connect to the server. The format of the message that it sends is as follows:

OFF	SIZ	DESC
0	1	code: REQ_CONNECT ('C')
1	1	protocol version := 0x01
2	1	transmit byte size: '7' or '8' bits
3	-	SIZE

This is what gets put into the the "payload" portion of the packet. All of the messages used in the protocol have an ASCII letter in the first byte that identifies what the message type is. Each request has an uppercase letter and each acknowledgement has the corresponding lowercase letter.

The connection-request message is fairly simple: it includes the protocol version number and the number of bits wide that the client thinks that the communication channel is. The version number is currently always 0x01 and is included for cross-compatibility with future versions of the protocol. The channel width is encoded into either a '7' or an '8' ASCII character. The client will think that the channel width is seven bits only if you tell it this on the command line.

When the server receives the connection request, it replies with the following message:

OFF	SIZ	DESC
0	1	code: ACK_CONNECT ('c')
1	1	protocol version := 0x01
2	1	transmit byte size: '7' or '8' bits
3	1	recommended request byte size: '7' or '8' bits
4	4	server maximum text-upload data size: H/M/M/L word
8	4	server maximum binary-upload data size: H/M/M/L word
12	4	server maximum text-download data size: H/M/M/L word
16	4	server maximum binary-download data size: H/M/M/L word
20	-	SIZE

The "protocol version" is what the server is using, currently always 0x01. The "transmit byte size" is the size that the user has specified on the command line that activated the server, and the "recommended request byte size" is a '7' if either the "transmit byte size" of the either the client or server is seven bits, or '8' otherwise. This is what should be used for the all subsequent messages that are exchanged.

The server's reply also includes the maximum packet sizes that it can handle for uploading and downloading binary and text files. The client then takes the "min" of the server's maximum packet sizes and its own, and uses the resulting maximum packet sizes for the rest of the file exchange session. The maximum packet sizes in the server's reply are all 32-bit unsigned integers that are stored from most-significant to least-significant bytes (big endian order). I picked big-endian order because that is the order used most commonly in inter-machine protocols.

The reason that the client doesn't have to inform the server of the client's maximum packet sizes in its connection message is that the maximum packet size to use is included with each request to get the next packet of a download file. It is sufficient that the client knows the full max-packet information. Really, the "transmit byte size" field isn't needed in the server reply message either, but I wanted the packet-size fields to be size-aligned.

After all of the file exchanging is completed, the client sends the following message to terminate the connection and return the server back to its command-line mode:

OFF	SIZ	DESC
0	1	code: REQ_DISCONNECT ('Q')
1	-	SIZE

When the server receives this request, it replies with:

OFF	SIZ	DESC
0	1	code: ACK_DISCONNECT ('q')
1	-	SIZE

And then exits like it should. Note that once the server exits, it cannot accept any more packets, since they would be sent to whatever command shell you use on your Unix system, and wouldn't do anything useful, so if the client sends the disconnect message but doesn't receive any reply, it will time out and tell the user that it couldn't disconnect cleanly from the server. This should be a rare occurrence. Anyway, what the user would do then is re-enter his terminal program and send Ctrl-X's at the server until it exits like it should have.

This arrangement allows us to avoid the famous(?) "two armies" problem that is inherent in disconnecting two connected processes: there is no "clean" way to do it. What systems like Z-Modem and Berkeley Sockets do is to have the server wait for a period of time that is longer than N times the timeout period of the client so that if there is a retransmission of the disconnection request, it likely that it will be received and processed correctly by the server. This is the reason (presumably) that Z-Modem does an annoying pause of 15 seconds or so after you finish transferring files. I think that my solution is much nicer, since the server can exit immediately (even though my server delays for 1 second, just so that your shell prompt will be cleanly in your modem's ARQ buffer when you re-enter your terminal program, if you have a hardware-flow-control modem).

4.2. FILE UPLOADING

Okay, so between connecting to and disconnecting from the server, actual

useful stuff happens, including uploading and downloading files. The uploading and downloading requests operate much like the regular file operations of open, close, read, and write. Really, the FX protocol makes the server program a special kind of file server.

When the client decides that it wants to upload a file, it first informs the server about this by sending the following message:

OFF	SIZ	DESC
0	1	code: REQ_UPLOAD_OPEN ('U')
1	1	data type: 't'=text file, 'b'=binary file: 'd'=directory
2	4	estimated file size: H/M/M/L word
6	2	permissions ("-----sgr:wxrwxrwx"), like Unix, H:L
8	12	modified date: BCD format: <YY:YY:MM:DD:hh:mm:ss:tt:tw:GG:gg:aa>
20	n	filename, null-terminated
20+n	-	SIZE

The "data type" field tells whether a text or binary file will be uploaded. There is a provision for "uploading" a directory entry (as part of uploading and downloading entire directory hierarchies), but support for this is not implemented yet. Also, it makes no difference to a Unix system whether a file contains text or binary data, but it may make a difference to other operating systems (like Mess-DOS). The "estimated file size" field isn't really used either, but it allows the server to make intelligent decisions about pre-allocating space, buffering, etc., if it needed to. However, it is currently not filled in by the client, since file-size information is difficult to extract from Commodore-DOS. The file size is an unsigned 32-bit quantity.

The permissions field is currently not supported by the server, but it is intended to allow file permissions to be preserved when passing files from one system to another. The interpretation of the 16 bits of this field is like it is with the Unix operating system: "rwx" bits for the owner, group, and other, and execute-as-owner, execute-as-group bits. The owner-id and group-id fields aren't included since they are generally not portable across systems, and even if they were, we usually want to receive files as our own owner-id and our own group-id.

The "modification date" field is not currently filled in either, since this information is even harder to come across with Commodore-DOS, but when it is, it will have a 12-byte BCD format. The "YY:YY:MM:DD:hh:mm:ss" sub-fields should be easy enough to figure out, and the "tt:t" fields contain thousandths of seconds. The "w" field contains the day of the week, coded as 0-6 for Sunday to Saturday, and 7 for "unknown". The "GG:gg" fields contain the number of hours and minutes that your time zone is off from GMT. If the number is negative (in the western hemisphere), then the regular positive number of hours will be used, execept that the 0x80 bit of the hours byte will be set. Finally, the "aa" sub-field is used to encode the accuracy of the timestamp. The way that it is interpreted is that the time value is accurate to plus/minus 2^aa milliseconds. For example, if my clock were accurate to within one second, then this field would be set to 10 in BCD (2^10 == 1024ms). A value of 99 means "unknown" (or that the clock could be off by many billions of billions of years).

I decided to go all out in defining the date field so that it will be useful in the future when "world consciousness" will be much more important than it is today.

And last but certainly not least, the filename is encoded in ASCII with a trailing zero byte.

Upon receiving this request, the server will attempt to create a file according to your specifications, and will send back a reply of the form:

OFF	SIZ	DESC
0	1	code: ACK_UPLOAD_OPEN ('u')
1	1	error code: 'y'=successful, 'n'=open unsuccessful
2	-	SIZE

The "error code" field tells whether the open operation was successful or not. If it was, then the client can continue with uploading its file; if not, then that file cannot be uploaded (and that the upload channel doesn't need to be closed). It's up to the client whether to go on to the next file, abort, or ask the user for help. The client will currently report an error to the user and then go onto the next file. Of course, it's likely that whatever caused the error in creating the current file will also cause an error in creating subsequent files (insufficient access permissions on the current directory, disk full, etc.). The server will overwrite any existing file with the same name (since asking permission, etc., would require extra mechanism, and would probably be a nuisance anyway).

If the upload channel is opened successfully, then the packets of upload data should be sent to the server one at a time, until all of the data is uploaded. The client sends the following message to the server to upload a packet of data:

OFF	SIZ	DESC
0	1	code; REQ_UPLOAD_PACKET ('R')
1	1	upload sequence number
2	4	data length: H/M/M/L word
6	n	data
6+n	-	SIZE

The "upload sequence number", which was described before, is used to make sure that retransmissions of packets are detected and handled properly, so that each packet of data only appears in the file once. The "data length" field tells the number of user data bytes that follow in the packet, and then the actual user data bytes appear. The "data length" field is actually redundant, but I figured that it would make programming a little easier, and allows additional error checking. Normally, each upload-data packet will contain the maximum-packet-size number of bytes of user data (according to whether text or binary data is being uploaded), except for the last packet, which will contain the number of data bytes that are left in the file. However, each packet is allowed to contain anywhere from 1 to the maximum-packet-size number of bytes: whatever the client wishes to use. Variable-sized packets are a Good Thing (TM, Pat. Pend.). You will note that the data-size values are also what will be used for the "read" and "write" system calls on the client and server, respectively. I/O will be done in big, efficient chunks.

Upon receiving each upload packet, the server replies with the following acknowledgement message:

OFF	SIZ	DESC
0	1	code: ACK_UPLOAD_PACKET ('r')
1	1	upload sequence number
2	-	SIZE

I don't think that the "sequence number" field is actually necessary here, but it is included to allow for future expansion and to provide redundancy for protocol-error checking.

When the client has uploaded all of the packets of the file currently being uploaded, it then sends the following message:

OFF	SIZ	DESC
0	1	code: REQ_UPLOAD_CLOSE ('V')
1	-	SIZE

This will close the upload channel and will finish writing the uploaded file to the Unix file system. The server will then respond with the following

message to acknowledge the request:

OFF	SIZ	DESC
0	1	code: ACK_UPLOAD_CLOSE ('v')
1	4	number of bytes uploaded: H/M/M/L word
5	-	SIZE

The "number of bytes" field is actually redundant, but is used for additional error checking.

4.3. FILE DOWNLOADING

Downloading files is analogous to uploading them: first we open the download channel/file, then we download the packets, and then we close the download channel.

To open the download channel, the client sends the following request to the server:

OFF	SIZ	DESC
0	1	code: REQ_DOWNLOAD_OPEN ('D')
1	-	SIZE

To which the server replies with:

OFF	SIZ	DESC
0	1	code: ACK_DOWNLOAD_OPEN ('d')
1	1	data type: '0'=no more files (eom),'t'=text,'b'=bin,'e'=err,'d'=dir
2	4	estimated file size: H/M/M/L word
6	2	permissions ("-----sgr:wxrwxrwx"), like Unix, H:L
8	12	modified date: BCD format: <YY:YY:MM:DD:hh:mm:ss:tt:tw:GG:gg:aa>
20	n	filename, null-terminated
20+n	-	SIZE

The file information is the same as for opening an upload file, except that there are more possible return conditions, and all of the "meta data" fields are actually filled in by the Unix host (since this information is actually conveniently available via the "stat" system call).

If the server replies with a '0' "data type" code, then this means that the server has no more files to offer for downloading. The filenames to download are taken one at a time, from left to right, from the command line that was used to start the server. When the server runs out, then the downloading session is complete and the client disconnects (since the client uploads its files first).

Alternatively, the server could reply with a 'e' code, which means that it could not open the next filename given on its command line. An error return is generated so that the client can inform the user that the file could not be downloaded. This will normally result from the user giving a bad filename on the command line. The client will continue the downloading process by closing the download channel (below) asking for the next file by re-opening the download channel. The download channel needs to be closed on this condition since otherwise there would be no way of distinguishing retransmissions from new requests at the server.

Finally, the server can reply with a 't' or 'b' code ('d' for directories is not currently implemented) indicating that the file was correctly opened and is either text or binary (as specified on the server's command line). Of the meta information about the file, only the filename and file size are currently used: the file is named according to the given name, translated to PETSCII and truncated to 16 characters, and the file size is reported to the user so that he can monitor downloading progress. I am not sure what to do yet about name collisions on the Commodore end: either ask the user whether to overwrite the file, automatically overwrite the file anyway, or automatically give the file a slightly different name and download normally. I think that for the time being, I will just overwrite the existing file. This will mean that you'll want to be extra careful in putting the filenames onto the correct command line (the client's or the server's), although there won't be a problem if the file doesn't exist on the machine whose command line you put the name on.

When the file handling is all squared away and the download channel is opened, the client then sucks packets out of the file until the end of the file is reached. The packets are sucked out with the following request:

OFF	SIZ	DESC
0	1	code: REQ DOWNLOAD PACKET ('S')

```
  1     1   download sequence number
  2     4   maximum acceptable data length: H/M/M/L word
  6     -   SIZE
```

The "download sequence number" is used to distinguish retransmissions from requests for new packets, and the client tells the server the "maximum acceptable data length" for the reply packet. Although the max-packet information is actually static during the connection, I included it here in every "read" request since I didn't really want the server to keep that particular bit of "state" internally.

The server replies to the download-packet request with the following message:

```
OFF   SIZ   DESC
---   ---   -----
  0     1   code: ACK_DOWNLOAD_PACKET ('s')
  1     1   download sequence number
  2     4   data length: H/M/M/L word, 0==EOF
  6     n   data
6+n     -   SIZE
```

This is the only "large" message that the server can produce. It includes the sequence number, the number of bytes that are actually included, and the user data. The number of data bytes in the packet is allowed to be smaller than the number of bytes requested, but this is normally only the case for the last packet of the file.

To indicate that the end of file has been reached and that no more user data is available, the server will return a download packet with zero bytes of user data in it. Upon receiving this, the client will close the download channel with the following message:

```
OFF   SIZ   DESC
---   ---   -----
  0     1   code: REQ_DOWNLOAD_CLOSE ('E')
  1     -   SIZE
```

And the server will reply with:

```
OFF   SIZ   DESC
---   ---   -----
  0     1   code: ACK_DOWNLOAD_CLOSE ('e')
  1     4   number of file bytes downloaded: H/M/M/L word
  5     -   SIZE
```

The "number of file bytes downloaded" field is redundant but included for additional error checking. After closing a file, the client will then ask for the next file, or will disconnect if the last file to download was just closed.

4.4. ERROR HANDLING

With all of the server calls except for disconnecting (discussed earlier), the is the possibility that either the request message from the client or the reply message from the server will become garbled and be dropped by the packet-delivery layer of the software. To recover from this, if the client detects an extended period of inactivity on the serial line for received data (where "extended period" is defined as being "about five seconds"), then the client will assume that something went wrong and it will retransmit the request.

As pointed out way above, there are two possible reasons for a retransmission being needed: either the request packet was corrupted and dropped, or the reply packet was corrupted and dropped. In the format case, the request wasn't processed by the server, but in the latter case, it was. Since we don't want the server to perform an file operation twice (this is really what the six file-transfer client operations really boil down to from the server's perspective), the server must keep four pieces of internal state: the last upload sequence number, the last download sequence number, whether the upload file is open, and whether the download file is open.

If an upload-open request is received and the file to be uploaded is not open, the the request must be a new one and the server processes it and sends back a reply like normal. If an upload-open request is receive and the upload file IS currently open, then it must be the case that the current request is a retransmission, so all theat the server needs to do is to give a positive reply without performing any internal file operations. The same holds true for the download-open call and for both of the close calls (except that the operation has already been processed if the file is CLOSED).

For the packet-upload and packet-download requests, sequence numbers are used

to detect duplicates. You will note that these sequence numbers are distinct from one another, and, in fact, that the entire upload and download file-transfer channels are distinct and independent from each another. This is to allow for the future possibility of simultaneous file uploading and downloading. In fact, if stream numbers (file descriptors) were added to the open/read/write/close requests, then we could have us a full-blown remote-host over-the-phone interactive file server. But anywho, sequence numbers start from 0x00 for the first packet transferred and increment modulo 256 from there.

Note that for high-speed data-compression modems (like I have) that already include error detection and recovery at a level hidden from the user, the FX protocol will work particularly well: there will never be an error, never be a timeout delay, and never be a retransmission. And, really, the CRC-32 error computation and checking is pretty much a zero cost. But, if something does go wrong, outside of the modem-to-modem connection, the FX protocol is right there to pick up the pieces and carry on.

6. CONCLUSION

You'll have to wait to get your hands on the program. The Unix Server program is almost 100% (except for a few design changes that I made while writing this document), and the ACE program is implemented except for the error handling and text conversion. Both programs will be released with the next release of ACE, which will be Real Soon Now (TM).

Here is my performance testing so far, using my USR Sportster modem over a 14.4-kbps phone connection, with a 38.4-kbps link to my modem from my C128, to my usual Unix host:

Using FX to/from the ACE ramdisk, REU:

```
Download 156,260 bytes of ~text:        time= 54.1 sec, rate=2888 cps.
Download 151,267 bytes of tabular text: time= 45.9 sec, rate=3296 cps.
Download 141,299 bytes of JPEG image:   time= 92.5 sec, rate=1528 cps.
Upload   156,260 bytes of ~text:        time= 57.4 sec, rate=2722 cps.
Upload   151,267 bytes of tabular text: time= 45.3 sec, rate=3339 cps.
Upload   141,299 bytes of JPEG image:   time= 95.0 sec, rate=1487 cps.
```

Using FX to/from my CMD Hard Drive:

```
Download 156,260 bytes of ~text:        time= 83.4 sec, rate=1874 cps.
Download 151,267 bytes of tabular text: time= 75.4 sec, rate=2006 cps.
Download 141,299 bytes of JPEG image:   time=118.2 sec, rate=1195 cps.
Upload   156,260 bytes of ~text:        time= 77.9 sec, rate=2006 cps.
Upload   151,267 bytes of tabular text: time= 66.2 sec, rate=2285 cps.
Upload   141,299 bytes of JPEG image:   time=114.2 sec, rate=1237 cps.
```

Using DesTerm-128 v2.00 to/from my CMD Hard Drive, Y-Modem:

```
Download 156,260 bytes of ~text:        time=189.5 sec, rate= 824 cps.
Download 151,267 bytes of tabular text: time=180.4 sec, rate= 839 cps.
Download 141,299 bytes of JPEG image:   time=199.9 sec, rate= 707 cps.
Upload   156,260 bytes of ~text:        time=255.1 sec, rate= 611 cps.
Upload   151,267 bytes of tabular text: time=238.6 sec, rate= 634 cps.
Upload   141,299 bytes of JPEG image:   time=233.0 sec, rate= 606 cps.
```

Using NovaTerm-64 v9.5 to my CMD Hard Drive, Z-Modem, C64 mode:

```
Download 156,260 bytes of ~text:        time=245.8 sec, rate= 636 cps.
Download 151,267 bytes of tabular text: time=230.0 sec, rate= 658 cps.
Download 141,299 bytes of JPEG image:   time=262.6 sec, rate= 538 cps.
```

(There is no Z-Modem uploading support)

So there you have it: my simple protocol blows the others away. QED.

==========

DESIGN AND IMPLEMENTATION OF A 'REAL' OPERATING SYSTEM FOR THE 128: PART II

by Craig S. Bruce <csbruce@ccnga.uwaterloo.ca>

0. PREFACE

There has been a slight change in plans. I originally intended this article to give the design of a theoretical distributed multitasking microkernel operating system for the C128. I have decided to go a different route: to take out the distributed component for now and implement a real multitasking microkernel OS for a single machine and extend the system to be distributed later. The implementation so far is, of course, only in the prototype stage and the application for it is only a demo. Part III of this series will extend this demo system into, perhaps, a usable distributed operating system.

1. INTRODUCTION

The previous article talked about the general approach to building a multitasking microkernel OS for the C128. It is assumed here that you have read and understood the previous article. This article goes into the grungy details of implementing such a beast. The prototype kernel implementation provides system calls to create and "exit" user processes, obtain status information, delay execution of a process for a specified period of time, and to perform message-passing interprocess communication.

Currently, there is no real memory management, no real device drivers, and no process-resource reclamation. More "infrastructure" features need to be added before a command-shell environment or any such thing could be supported, though not toooo many more; the Commodore-Kernal Server in the demo system makes the $FFD2 (CHROUT) routine of the Commodore Kernal available to all other processes in the demo system. It could easily be modified to provide all of the Commodore-Kernal features to the other processes, thereby giving us a basic I/O sub-system.

There is also no way to dynamically load external programs, so the test programs have to be assembled with the kernel code. Loading external programs in this type of environment has the requirement that the program will have to be relocated upon being loaded to an address that would only be known at load time. There are two ways to go on this: load all programs to a fixed address or load them to dynamic addresses. If you load them to a fixed address, then you can only have one processes loaded and concurrently running on each bank of internal memory of the C128 and then demand-swap all of the other processes into and out of these (two) slots, presumably from REU memory (any other type would be much too slow). IHMO, even with REU memory, this would be too slow, especially for a microkernel environment. So, programs will need to be loaded to dynamic addresses. Fortunately, I have a program-relocation mechanism in the works.

The entire kernel and the demo program fits in C128 memory in the slot between $1300 and $1BFF of RAM0. The kernel uses storage from $C00-$CFF and $2000-$BFFF. The latter section of memory is used up in 768-byte chunks by each new process, so there can be a total of 53 concurrently executing user processes in the system.

2. TEST PROGRAM

The test program includes no provisions for user interaction, so it may not be something that you can impress your friends with, but I can assure you that all kinds multitasking stuff is going on behind the scenes to make everything happen.

The demo test program creates ten processes. There are five "delay" processes, two "blabbering" processes, a Commodore-Kernal-Server process, one SID-banging process, and the Null process. (Note that when I use the word "kernel" I am referring to the OS that I have written, and when I use the word "Kernal", I am referring to the Commodore Kernal in ROM).

The purpose of the Commodore-Kernal Server is to receive requests from the worker processes to call the CHROUT routine to print a given line of text out to the screen. The Kernal server is the only processes that is allowed to call the Commodore-Kernal routines. Since it can only process one request from a client process at a time, calls to the Commodore Kernal are effectively "serialized" (made to happen one after another in time), which is good, since things would blow up pretty badly if two accesses the Commodore Kernal were to happen concurrently.

The "delay" processes, numbered from 1 to 5, each delay for a period of N seconds and then request the Kernal Server to print a "I'm alive" message to the screen. The number N of seconds to delay is the number of the process. You should be able to observe from watching the execution that each delay process prints a message to the screen with approximately the correct period between messages. Note that while these processes are delaying, they don't use any CPU time, so the CPU time is allocated to other processes. If you try holding down the C= key to slow the scrolling or run the system in Slow mode, you will still notice that the delay processes generate their output at approximately the right time.

The two "blabbering" processes, named "blabber" and "spinner", continually send print messages to the Commodore-Server process. You will observe that the messages from each comes pretty much prefectly interleaved with each other, and with the output of the delay process, because of the interprocess communication scheduling policy: FIFO (first-in, first-out).

The SID-banging process runs continuously in the background. It increments the 16-bit frequency of voice #1 from $0000 to $ffff and then suspends its

execution for two seconds and then repeats. A square wave with even pulse widths is used. When the SID process delays for two seconds, you will notice that the printing operation of the other processes speeds up a bit. This is because the SID process is a heavy CPU user while it is active. The amount of slow-down of the rest of the system is limited a bit because the SID process runs with a slightly lower priority than the other processes in the system (which makes the sound increment slightly slower than it otherwise would -- there's only so much CPU to go around).

The Null process is not actually needed in this exercise, but it would normally be used to insure that the system always had some process to schedule. All that it does is increment the binary value in locations $0400-$0403 (on the 40-column screen) whenever it is active. It has the lowest priority in the system and never gets to execute unless all of the other processes are blocked (i.e., are suspended for some reason).

When you get tired of watching the demo, you can just hit the RESTORE key. This will cause an NMI which will make the system exit back to BASIC. The system does not have the ability to handle external events (like key strokes) at this time. A couple of locations on the 40-column screen are used for status information, so you will want to run the demo on the 80-column screen.

3. PROCESS CONTROL

A process is a user program that is in an active state of execution. A process is periodically given a certain amount of CPU time to execute its code, and then CPU attention is taken away from it to execute other processes. This may sound like you're simply making N processes run N times slower, and this is true in the worst case, but the normal case is that many processes in the system will be blocked (for whatever reason) and will not require any more CPU time until they wake up again (for whatever reason). Therefore, multitasking is a "winnable" proposition.

In our system, the process that the CPU is currently executing is changed every 1/60 of a second. This is a convenient "quantum" period for a number of reasons, including the fact that, thanks to the MMU of the C128, "context switching" can be efficiently performed this quickly.

3.1. PROCESS-CONTROL CALLS

There are six kernel calls that deal with process control:

CALL NAME	INPUT ARGUMENTS	RETURN VALUES
Create	(.AY=address, .X=priority)	.AY=newPid, .CS=err(.A=errcode)
Exit	(.A=code, .X=$00)	<none>
MyPid	( )	.AY=pid
MyParentPid	( )	.AY=parentPid
Suspend	( )	<none>
Delay	(.AY=jiffies)	<none>

3.1.1. CREATE

The Create() kernel call is used to create a new process. The first input argument is the code address, and it is passed in the .AY register (.A is loaded with the low byte of the address, and the .Y register is loaded with the high byte of the address--.AY for short). The code must be present in memory and be ready to be executed, since there is no facility for loading external programs. Also, if the code is not re-entrant, then there must be no other process already executing it or things will likely blow up. Re-entrant code is code that can be executed by multiple processes simultaneously without conflicts, essentially because there are no global variables that could be banged on by more than one process at a time.

The priority argument is the priority to execute the new process at. Valid values for this argument are on the range 0 to 127. The system keeps a list at all times of all the processes that are ready to execute, called the "ready list". The way that the scheduling works is that a pointer to the "active" process is kept (the one that is currently executing) and this pointer cycles through the ready list, trying to activate each process in turn, every 1/60 of a second. This is roundrobin scheduling. The priority of a process determines the number of cycles that the active-process pointer has to take through the list before the process is activated. So, if a process has priority 1, then it will be activated on every round; if it has a priority of 2, then it will be activated on every second round; and if it has a priority of 86, then it will be activated only on every 86th round through the ready list. The higher the priority value, the slower the process executes. This policy gives a fair allocation of the CPU to the various processes in the system.

Normally, foreground processes (ones that perform actions right in front of

the user's face) should have a priority of 1, and background processes should have a relatively lower priority. A priority value of 0 for a process means that when the process is activated, it will not be deactivated again until it blocks for some reason. This priority level should be reserved for urgent computations that block often, since it has the potential to starve out the rest of the system. The Null process executes at a special priority level (255) that makes it so that it will only be activated if there are no other processes in the ready list.

The Create() call returns with the carry flag clear and the process id of the newly created process in the .AY register upon success, or returns with the carry flag set and an error code in the .A register upon failure. I do not have the complete list of error conditions figured out a this time, but errors will usually happen on a call like this because of a lack of resources (memory) for the kernel's internal data structures. Upon successful return, the child process is created, made ready, and may be activated by the system at any time. The first instruction to be executed by the child will be at the value given for the code address.

The newly created process will have a clear individual stack, except for a couple of bytes on the very bottom of it (high addresses), and a clear individual zeropage. Processes are allowed to make full use of every location in their zeropage, except for the I/O registers at locations 0 and 1, and full use of their stack, except that they must make sure that about a dozen bytes are available on the stack at all times in case an interrupt happens.

3.1.2. EXIT

The Exit() kernel call is used to remove the current process from the system. There are two input arguments: the .A register contains the return code that will be made available to the parent process if it is interested (though not in the current implementation), and a value in the .X register, which must currently be $00. I haven't figured out exactly how the exit mechanism should work yet and it currently only has a minimal implementation. The call does make the kernel reclaim the resources that were allocated to the process yet, although this functionality will be needed in any real operating system.

There is no return value from the Exit() call, because the call never returns. The semantics of the call is the the process calling Exit() will never be made active again. All processes should call Exit() when they are finished executing, or they can achieve the same result by executing an RTS instruction at the end of their main routine. The kernel pushes the address of an Exit() stub routine onto the top of the stack of a user process when it is created, and the user process will exit with a return code of $00 in this case.

3.1.3. MY_PID

The MyPid() kernel call is used to return the process identifier of the current process. This call is very simple, takes no arguments, and executes very quickly. The return value is the process id of the calling process and is returned in the .AY register. This call cannot fail, because the current process must exist in order to make the call in the first place, so there is no error-return condition.

3.1.4. MY_PARENT_PID

The MyParentPid() kernel call is used to return the process identifier of the parent process of the current process (i.e., the process that created the current process). This call is simple, takes no arguments, and executes quickly, very much like the MyPid() call. No error returns are possible, and the process id of the parent to the current process is returned in the .AY register. But note: it is not guaranteed that the parent process will still exist either before or after the current process makes this call; it may have Exit()ed. I may re-think this semantic.

This call is useful for setting up interprocess communication between a child process and its parent.

3.1.5. SUSPEND

The Suspend() kernel call is used to suspend the execution of the currently executing process for an indefinite period of time. Currently, this period of time is forever, since there is no corresponding "Resume" system call that another process can call in order to wake up the process that suspended itself. The reason that this call is made available is because the guts of what it does is required by other kernel operations, and the cost of making this call user-accessible was three 6502 instructions. This call may be retracted in the future, since it may cause programmers to do bad things.

The call takes no arguments, returns no values, and currently, will never return at all, much like Exit().

3.1.6. DELAY

The Delay() kernel call is used to suspend the execution of the current process for a user-specified period of time. The delay period is given in units of jiffies (1/60ths of a second). The unsigned 16-bit delay period is passed in in the .AY registers, giving a maximum possible delay period of about 18 minutes. If a user process requests to delay for a period of zero jiffies, its execution will not be suspended at all and the Delay() primitive will return immediately.

Since there may be other processes in the system doing things when the current processes wakes up after doing a delay, you can think of the process delaying for "at least" the period of time that you specify. Actually, to muddy things even more, your process will always go to sleep at a moment in time that is inbetween two ticks of the jiffy clock, so the first "jiffy" that your process waits may actually be any period between a couple of microseconds to almost a full jiffy, with a statistical average of half a jiffy. This is an artifact of any coarse-tick-based mechanism.

To muddy things again, the jiffy ticks, which are currently based on VIC raster interrupts (one per screen update), may not be processed immediately when they occur, since the IRQ may be delayed by a small period of time if interrupts are disabled in the processor status register when the jiffy tick happens. And finally, you should note that you will have a difficult time using this call for true "real time" periodic operations, like performing some specific task precisely every tenth of a second, since the call specifies a period to delay for, rather than a time to wake up at. The actual period of your process' activations will be determined by the waiting time plus the time skew caused by the processing that your process does. A DelayUntil() call easily could be implemented, if I figure that it will be needed for anything.

Currently, the scheduling policy is to make processes active immediately after they are awakened, so this makes the activities of other processes less of a worry to accurate timing. Unix does a similar thing by giving a freshly awakened process a temporarily high priority, since it is probably likely that the process will do some small think and then block again. This policy statistically improves concurrency.

3.2. PROCESS CONTROL BLOCKS

A Process Control Block (PCB) is the data structure that the kernel keeps the information that it needs to know about a process in. A Process identifier (pid) is actually the RAM0 address of the process control block of a process, for convenience, though this will have to change later. The fields of the process control block are shown here, organized into classes:

OFF	SIZ	CLASS	LABEL	
0	2	queue	pcbNext	
2	2	queue	pcbPrev	
4	1	queue	pcbIsHead	
5	1	queue	pcbQCount	
6	1	ctxt	pcbSP	
7	1	ctxt	pcbStackPage	
8	1	ctxt	pcbZeroPage	
9	1	ctxt	pcbD506	
10	1	sched	pcbPriority	
11	1	sched	pcbCountdown	
12	2	sched	pcbWakeupTime	
12	2	ipc	pcbSendMsgPtr	(overlap)
12	2	ipc	pcbRecvMsgPtr	(overlap)
14	2	ipc/q	pcbSendQHead	
16	2	ipc/q	pcbSendQTail	
18	1	ipc/q	pcbSendQFlag	
19	1	ipc/q	pcbSendQCount	
20	2	ipc	pcbBlockedOn	
22	2	ipc	pcbReceiveFrom	
24	2	proc	pcbParent	
26	1	proc	pcbState	
27	-	-	SIZE	

3.2.1. QUEUE-CLASS FIELDS

The first four fields, of the class "queue" are used for maintaining a process control block in queues with other PCBs. Some general-purpose queue-handling routines have been written to make queue management easier: QueueInit(), QueueInsert(), and QueueUnlink(). Each queue has a head node, and the nodes in a doubly linked circular order. This means that each node in the queue has a forward ("pcbNext") and a backward ("pcbPrev") pointer and that the first node points back to the head and the last node in a list points forward to the

head. This organization removes all of the quirks of handling null pointers from the code. Using a doubly linked organization makes it easy to remove an arbitrary node from the middle of a queue.

Each node also has a "pcbIsHead" field which is always False (zero) and a "pcbQCount" field which is always zero. The head is the same as an entry in the queue, except that its "pcbIsHead" field is set to True ($ff) and its "pcbQCount" field records the number of nodes that are in the queue at any time. The "pcbIsHead" field is checked when scanning a list to tell if you've bumped back into the head node again, indicating the end of the list. The "pcbQCount" field is very convenient to check to see whether the queue is empty or not.

All of the processes that are ready to execute in the system are kept in the ready queue. The PCB of the Null process acts as the head for this queue, and is also an active node in the queue (a small but harmless kluge). The pointer to the active process is kept in a kernel-zero-page variable, and sweeps through the circularly linked ready-process list to activate new processes. The active PCB is not removed from the ready list while it is active.

3.2.2. CONTEXT-CLASS FIELDS

The next four fields, of the class "ctxt", store the "context" of a process that is not stored on the process' stack when it is not executing. These fields include space for the stack pointer, the stack page, the zeropage, and the contents of the MMU register at location $d506. The stack pointer is what was in the SP register of the CPU when the process last paused. The stack page and the zeropage values are the values in MMU registers $d505 and $d507, respectively; these are the page numbers of the pages in RAM0 memory that are allocated to a process. These pages can only be in RAM0 unless common memory is disabled, for hardware reasons. I may allow these pages to be in either RAM0 or RAM1 if there is a need later. The $d506 register of the MMU stores the most-significant bits of the RAM bank that is selected if you have expanded internal memory on your 128 (a la TwinCities-128) and the bank selection for REU (DMA) operations.

The rest of a process' context is stored on its stack. Here is what a process' stack looks like just after it has been created:

ADDR	SP-REL	DESCRIPTION
$ff	sp+09	exitaddr-1.h
$fe	sp+08	exitaddr-1.l
$fd	sp+07	pc.h
$fc	sp+06	pc.l
$fb	sp+05	status register
$fa	sp+04	.A
$f9	sp+03	.X
$f8	sp+02	.Y
$f7	sp+01	$ff00 save
$f6	sp+00	-empty-

The "exitaddr" is the address of the routine in the kernel that will terminate a process if it executes an RTS from its main routine. The address is in the regular low-high order (although it is pushed on high-low since the stack grows downward in memory) but the value pushed is actually one less than the address of the routine, because this is what JSR pushes onto the stack and this is what RTS expects to find. The "pc" low and high fields give the address of the next instruction to be executed by the process when it is activated. When the process is first created, this will be the address of the first instruction. The "pc" value is the actual address, not one before, because this is what a hardware interrupt pushes onto the stack, and this is what the RTI instruction expects to find.

The "status register", ".A", ".X", and ".Y" fields contain the values to be loaded into the corresponding registers inside of the CPU when the process is activated. For a new process, these values are all zero.

The "$ff00 save" is the value to be loaded into the $ff00 "shadow" register of the MMU when the process is activated. This gives the memory context that the process is to execute in. As the kernel currently only works with one bank configuration (RAM0, NO BASIC ROM, I/O enabled, KERNAL ROM enabled), this is the value put here when a process is created.

The final field is "-empty-" because the stack pointer in the 6502 points to the next location in stack memory that will be used. All other values in the stack are relative to this. Upon startup, the stack-pointer field of the process control block will be set to $f6, which is what the table above shows.

The stack contents look exactly the same after a process has been interrupted

by a hardware interrupt, except that the stack pointer will likely be lower in the stack memory, so the absolute addresses in stack memory in the above table no longer apply and the "exitaddr" bytes are not part of the interrupt context. That things look the same is no coincidence; on startup, we set up the stack to make things look as if an interrupt had just occurred, and to start a process executing, we execute the code that returns from an interrupt, which loads the "context" that is on the stack into the process registers, and we are ready to rock.

The above stack organization is exactly the same as it is for processing interrupts normally using the Commodore-Kernal environment on the 128, and this too is no coincidence because the code that sets up the stack like this upon an interrupt is burned into the Kernal ROM and there is very little that I can do about it. Fortunately, the organization is just fine for our purposes.

3.2.3. SCHEDULE-CLASS FIELDS

The next three fields, of class "sched", are used to schedule the process. The "pcbPriority" field gives the relative priority of the process according to the scheme already discussed. The "pcbCountdown" field is used to keep count of the number of remaining times that the process will have to be bypassed in cycling through the ready queue before the process will be activated again. When a process gives up the CPU upon the expiration of its time quantum, the "pcbCountdown" field is loaded with the pcbPriority of the process. When the "pcbCountdown" value reaches zero, the process is selected for activation.

The "pcbWakeupTime" field is used with the Delay() kernel call to indicate the absolute system time when the process should be activated again. The current time in the system is kept in a 16-bit kernel variable, and wraps around every 18.2 minutes. If the process is not currently time-delayed, then the WakeupTime field is not used (in fact, the memory may be used to record other status information).

3.2.4. IPC-CLASS FIELDS

The eight eight fields, of classes "ipc" and "ipc/q" are used for interprocess communication (message passing). The first two fields, "pcbSendMsgPtr" and "pcbRecvMsgPtr", are used for storing temporary values for handling message requests; the four "ipc/q"-class fields are used to implement the head of a queue of processes that are waiting to communicate with the current process, and the "pcbBlockedOn" field indicates which process this process is waiting to communicate with, if the current process is waiting. The first two fields actually overlap with each other and with the "pcbWakeupTime" field discussed earlier. This is okay since none of the fields will store active status information at the same time. The last field, "pcbReceiveFrom" is not used at this time, but will be used in the future for an primitive to receive a message only from a specific process. The interprocess communication is discussed in much greater detail later.

3.2.5. PROC-CLASS FIELDS

The final two fields, of class "proc", store information about the status of the process. I guess the same can be said of all the other fields. Anyway, the "pcbParent" field indicates which process is the process that created the current process, and the return value for the MyParentPid() kernel call is taken from this field.

The "pcbState" field gives the current state of the process. Here are the different possible process states:

STATE NAME	CODE
STATE_READY	$c0
STATE_SEND	$c1
STATE_RECEIVE	$c2
STATE_REPLY	$c3
STATE_DELAY	$c4
STATE_SUSPENDED	$c5

The STATE_READY state means that the process is in the ready queue. The STATE_SEND, STATE_RECEIVE, and STATE_REPLY states mean that a process is waiting for some interprocess communication primitive to be called by the process that it is communicating with. The STATE_DELAY state means that the process has called the Delay() primitive and is waiting for some period of real time to pass before it can be activated again. The STATE_SUSPENDED state means that a process has called the Suspend() or Exit() primitive and will never be made active again (currently, Suspend() and Exit() mean the same thing).

The state information is needed for some operations, and will definitely be needed by a Kill()-process operation, to find out what state a process is in so that it can be removed from any queue or whatever it is in, in order to obliterate all information about the process from the system. Currently, there is no Kill() call.

3.3. TASK CREATION

When the user calls for the creation of a new process in the system, lots of stuff has to happen. First, memory for the process control block, zero page, and stack page must be allocated. Currently, this allocation is performed very simply, by keeping a page pointer and incrementing it every time a page is allocated. The process control block ends up getting allocated 256 bytes even though it actually requires much less than that. Thus, each new process chews up 768 bytes of memory space (plus code). Also, there is currently no mechanism for recovering the memory allocated to a process, which is okay since the mechanism for Exit()ing a process is incomplete too.

The address that a PCB gets allocated at is used for the process' PID (process identifier). This is particularly useful since the real purpose of a PID is to conveniently locate the process control block. This will have to change in the future, however, since the PIDs will also have to locate the machine that a process is on.

Then the process control block must be initialized. It is initialized as follows:

FIELD	SIZ	CLASS	INITIAL VALUE
pcbNext	2	queue	0
pcbPrev	2	queue	0
pcbIsHead	1	queue	0
pcbQCount	1	queue	0
pcbSP	1	ctxt	$f6
pcbStackPage	1	ctxt	set to newly allocated space
pcbZeroPage	1	ctxt	set to newly allocated space
pcbD506	1	ctxt	$04
pcbPriority	1	sched	set to the given argument
pcbCountdown	1	sched	set to the same value as "pcbPriority"
pcbWakeupTime	2	sched	0 (overloaded field)
pcbSendQHead	2	ipc/q	set by the QueueInit() function for empty queue
pcbSendQTail	2	ipc/q	set by the QueueInit() function for empty queue
pcbSendQFlag	1	ipc/q	set by the QueueInit() function for empty queue
pcbSendQCount	1	ipc/q	set by the QueueInit() function for empty queue
pcbBlockedOn	2	ipc	0
pcbReceiveFrom	2	ipc	0
pcbParent	2	proc	set to the id of the creator process
pcbState	1	proc	STATE_SUSPENDED

The values on the top of the stack page are also initialized for the new process, as follows:

ADDR	SP-REL	DESCRIPTION	INITIAL VALUE
$ff	sp+09	exitaddr-1.h	high byte of ExitAddr-1: the exit routine
$fe	sp+08	exitaddr-1.l	low byte of ExitAddr-1: the exit routine
$fd	sp+07	pc.h	high byte of the code-execution address
$fc	sp+06	pc.l	high byte of the code-execution address
$fb	sp+05	status register	$00
$fa	sp+04	.A	$00
$f9	sp+03	.X	$00
$f8	sp+02	.Y	$00
$f7	sp+01	$ff00 save	$0e (Kernal ROM, I/O, rest RAM0)
$f6	sp+00	-empty-	--

And now, the new process is ready for action. We insert it into the ready queue in the next position after the current process, set its state to STATE_READY (actually, both of these operations are performed by the MakeReady() function, which is generally useful and is called from a number of places) and then we exit back to the calling process, returning the process id of the calling process. I should change this a little bit in the future, to make it exit to the newly created child process if the priority of the child process is greater than the priority of the parent.

3.4. CONTEXT SWITCHING

Context switching describes the procedure of switching control of the processor from a user process to the kernel and then switching control back to a user process. Normally, there is only one "style" of context switching in a system, but for a couple of design reasons, BOS actually has three "styles" of context switching: IRQ switching, JSR switching, and quick JSR switching.

IRQ-style switching is the one type normally implemented in operating systems for other architectures, so it will be the one that we cover first.

IRQ-style context switching involves saving the full context of a process onto its stack and into its process control block, switching into the kernel, doing work, switching back out of the kernel, and reloading the full context of a user process and activating to it. All of the work of saving and restoring the the stack portion of a process' context is handled by the ROM routines for IRQ (and NMI and BRK) handling. All we have to do is locate the current process control block, save the zero-page, stack-page, stack-pointer, and $d506 registers into the PCB, and load a $00 into the zero-page MMU register to switch to the kernel's zeropage (where some of the kernel's variables are stored). Note that the interrupt will be executed using the user process' stack; therefore, enough space should always be available on user stacks to handle this system overhead.

When we are done processing the interrupt, we execute the priority-management algorithm that was described earlier to select the next process to activate, and then restore the zero-page, stack-page, stack-pointer, and $d506 registers and execute the ROM stack-handling code for exiting from an interrupt. Note that there's a chance that we might well be exiting to a different user process from the one that was active when the interrupt occurred. There aren't many registers to save and restore, so context switching has a fairly low overhead, so there is no problem in doing it (at least) sixty times a second.

JSR-style context switching is pretty much the same as IRQ-style context switching, except that the stack will not have most of the processor registers already saved on it; it will only have the return address that performed the JSR. Immediately upon entering the kernel, interrupts are disabled to prevent all sorts of bad things from happening. Then a function is called, EnterKernel(), which will pull the return address of the process that called the JSR off the stack and increment it by one (since we will be exiting by using an RTI instruction rather than an RTS) and saves the other processor registers onto the stack in the same way that the interrupt-handling code in ROM would. Then we save the four additional registers into the PCB as before, activate the kernel zeropage, and we are switched in.

This style of context switching is used for kernel calls that will cause the calling process to block (like a non-zero Delay()). It would have been possible to organize the kernel calls to be entered by executing a BRK instruction, which would have caused the stack to be already set up in the same way as with IRQ interrupts, but I decided against this for two reasons: efficiency, it would have been slower to do this, and debugging (security?), since I only want the BRK condition to signal a bug in the code. The exit from this type of context switch is the same as for the IRQ style of context switch, since things are rigged to end up looking the same on the stack. This is a good thing, since the action that will cause a Delay()ed process to be re-activated will, in fact, be an IRQ interrupt.

Quick JSR-style context switching is used for kernel calls that will not block or cause a new process to be activated when they finish, such as MyPid() or (currently) Create(). No context has to be saved since the function will get in and out very quickly; all we have to do is switch to the kernel's zeropage and then switch back to the user's zeropage before exit.

There's one more note to make about return values. For the quick JSR-style context switch, there is no problem with return values, since we just have to load them into the processor registers and exit. With the full JSR-style context switch, the return values have to be put onto the user stack into the positions in the stack memory the hold the processor register contents, since these values will be what are restored into the processor immediately upon the return to the user process. There are no return values associated with the IRQ style of context switching (and there'd better not be), since an interrupt can happen at any point in the execution of a user process.

3.5. DELAY PRIMITIVE

There are two complementary halves to the implemention of the Delay() primitive: the half that is called by the user and causes a process to go to sleep, and the half that wakes up a sleeping process at the correct time. This latter half is executed by the 60-Hz system interrupt.

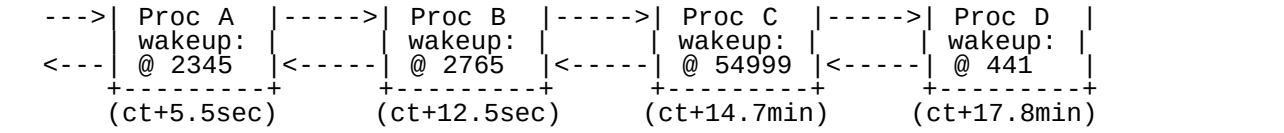
3.5.1. USER HALF OF THE DELAY PRIMITIVE

The first thing that the user half of the Delay() primitive does is check to see if the delay period is zero jiffies. If it is, then the primitive returns immediately to the calling process without rescheduling (without skipping to the next ready process in line). I may change this semantic, because it is often useful to have a primitive that yeilds process execution to the next ready process without actually blocking the current process.

If the delay period is longer than zero jiffies, then the current process is suspended and removed from the ready queue, and the absolute time that the process is to be reawakened is calculated and put into the "pcbWakeupTime" field of the PCB for the current process. The absolute wakeup time is calculated, of course, by adding the number of jiffies to delay to the current absolute time, which is maintained by the system and incremented on every (60 Hz) system interrupt.

Then the current process control block is inserted into the delay queue at the correct position. The delay queue is a queue (implemented in the standard way) of process control blocks for processes which are asleep, ordered by the absolute wakeup time of each process such that the process that will be awakened at the nearest time in the future is at the head of the list and that the process which will be awakened at the farthest point in the future is at the tail. The following diagram gives an example:

CurrentTime = 2016



There is a rub here: only 16 bits are used for storing times, which equals about 18.2 minutes, so we have to worry about time quantities overflowing and wrapping around. For example, if the current time is 48232 and a process wants to sleep for 18000 jiffies (5 minutes), then its wakeup time would be at 696 jiffies, accounting for the 16-bit wraparound, which is a lower numerical value than the current time, or than the wakeup time of any other process that will wake up before the current-time wraparound. In fact, all timers have this wraparound problem (although with 64-bit times, wraparound periods would be expressed in millions of millennia rather than in minutes). Sixteen bits is a good number of bits to use, however, because that is the maximum delay period (2^16-1).

When we insert a new process into the delay queue, we scan the delay queue from the head and continue until we find a record that has a time that is higher than or equal to the wakeup time of the new process (or we hit the end of the queue). Then, we insert the new process immediately before this point. To handle the wraparound problem, all comparisons of wakeup times are done using 17 bits (well, really 24 bits). For each value in the comparison, we add 65536 to it (set its 17th bit) if the value is less than the current time. We don't have to worry about the current time changing while we are doing this, because interrupts will be disabled for the entire time that we are executing the system call, as per usual. Things could go horribly wrong anyway if interrupts were not disabled.

Okay, so now our delaying process is removed from the ready queue, its complete context is saved, and it is put into the delay queue at the right spot. So, set the active process pointer to the next ready process in the system and finish by activating the next ready process.

3.5.2. SYSTEM HALF OF THE DELAY PRIMITIVE

During each 60-Hz system interrupt, the current time (jiffy counter) is incremented by one. Note that since this timer is only 16 bits wide, it is not suitable for keeping track of the current time of day; for this purpose, the TOD clocks in the CIA chips should (and will) be used. The jiffy counter may also be inaccurate if interrupts are disabled for a long period of time, such as they are during some Commodore-Kernal I/O operations.

After incrementing the time, the kernel checks to see if any Delay()ed processes need to be woken up. If there are no processes in the delay queue, then this is a quick check. If there are any processes in the queue, then if the wakeup time of the head process is equal to the current time, then that process is woken up and this check is performed repeatedly until the condition fails, since there may be multiple processes that want to be woken up at the same jiffy of absolute time. Note that because of the scheduling for a freshly unblocked process, the process that Delay()ed first will be the first one activated after it is woken up, if there are multiple processes woken up at the start of the same jiffy.

3.6. SYSTEM BOOTSTRAPPING

Operating systems always have a bootstrapping problem, because you always need to use the services of the operating system in order to start it up, but, of course, it's not started up yet, chicken and egg, catch-22. So, what usually ends up happening is that you just "fake it", start from somewhere, get the

ball rolling, and snowball up to a fully running system.

The first thing that the kernel does is change all of the interrupt vectors (IRQ, NMI, and BRK) to my custom routines. I need to cover all of the interrupts, since I chave the zero page during the execution of the system, and if a BRK or NMI were to happen and be serviced by the Commodore-Kernal ROM routines, all hell would break loose. Currently, the NMI and BRK routines just clean things up and return to BASIC.

Then we initialize the kernel variables, including the delay queue and the jiffy counter.

And then we fake the creation of the Null process. For the purposes of bootstrapping, the Null process doubles as the "Boot" process. Its process control block is not allocated in the normal way, either; it is at a fixed location, and its PCB doubles as the head of the process list. A kluge here and a hack there and the Null process is initialized and "joined in progress". Then, the Null process creates the Init process, using a standard call to the Create() primitive, and then the Null process goes into an endless loop of incrementing the 32-bit value at addresses $400-$403, the first four locations of the 40-column screen memory. It doesn't matter whether you run BOS with the clock in Fast or Slow mode, except in terms of performance.

It is the responsibility of the Init process to start up all of the user processes in the user application after Init starts running. In the current implementation, Init starts up all of the other processes in the test application and then becomes the Commodore-Kernal Server, which is a convenient organization, since all of the other processes can find out the pid of the Kernal Server merely by calling MyParentPid().

4. INTERPROCESS COMMUNICATION

In this system, processes are not strictly independent and competitive; many must cooperate and comunicate to get work done. To facilitiate this interprocess communication (IPC), a particular paradigm was chosen: the Remote Procedure Call (RPC) paradigm. RPC is a message-passing scheme that is used with the much-hyped Client/Server system-architecture model. Its operation parallels the implicit operations that take place when you call a local procedure (a subroutine).

The RPC message-passing paradigm is also coupled with a shared-memory paradigm to offer greater performance for passing around massive amounts of data. All processes in the system (and in the entire distributed system when this OS is extended) have global access to all of the memory in the system. The coupling of the two paradigms is such that you get the best of both worlds: the convenence and natural interprocess *coordination* (synchronization) semantics of RPC and the convenience and raw performance of shared storage.

4.1. MESSAGE-PASSING CALLS

The kernel provides three primitives for message passing:

CALL NAME	INPUT ARGUMENTS	RETURN VALUES
Send	(.AY=msgHead)	.CS=err(.A=errcode)
Receive	(.AY=msgHead)	.AY=senderPid
Reply	(.AY=msgHead[msgRet,msgData])	.CS=err(.A=errcode)

These calls will send a message from one process (the client) to another process (the server) and wait for a reply, receive a message from another process (a client), and reply to a message sent from another process (a client) that has been received, respectively.

4.1.1. MESSAGE-HEADER DATA STRUCTURE

Each of the message-passing primitives requires a pointer to a message-header data structure that is stored in the user program's data space. The message header must be initialized with appropriate values before a message can be sent. Note that this scheme of passing a pointer to a message header allows you to have multiple message headers lying around, initialized and ready for action, and you can easily pick between them. Here is what a message header looks like:

OFF	SIZ	CLASS	LABEL
0	2	pid	msgTo
2	2	pid	msgFrom
4	4	buf	msgBuf
8	2	buf	msgLen
10	4	buf	msgRepBuf
14	2	buf	msgRepLen

```
 16     1   data    msgOp
 17     1   data    msgRet
 18     2   data    msgObj
 20     4   data    msgData
 24     -   -       SIZE
```

You should not put too much faith in the offsets of the fields in the data structure remaining static; you should always use the label to access the fields of the structure, as in:

```
sta myMessageHeader+msgTo+0
sty myMessageHeader+msgTo+1
```

4.1.1.1. PID-CLASS FIELDS

The first two fields, of class "pid", are used to identify the processes involved in an RPC interaction. The "msgTo" field is the pid of the process that a message is to be/has been sent to, and the "pcbFrom" field is the id of the process which a message has been received from. For security reasons, the sender does not fill in the "pcbFrom" field; the kernel does after the message has been sent and the sender is blocked. (Or else the sender could fake being someone else). The "pcbTo" field is used as the destination for when a message is being sent and must be filled in with a legitimate value on a send operation, and the "pcbFrom" field is used as the destination when a message is being replied to, and must be filled in with a legitimate value on a reply operation. The "pcbTo" field is the only field of the message header that actually needs to have a legitimate value before a message can be sent.

4.1.1.1. BUF-CLASS FIELDS

The next four fields, of class "buf", point out the send and reply buffers in memory and the sizes of each. The send buffer ("msgBuf"/"msgLen") is expected to point to a region of near/far memory that contains valid data for a send operation, and the reply buffer ("msgRepBuf"/"msgRepLen") is expected to point to a valid area of memory for the server to fill in with any bulky result data from an RPC request. Each of the message-buffer pointers is four bytes in size to allow for future expansion when the kernel will support "far" memory that will be accessed through 32-bit pointers. User processes are expected to access these "far" buffers directly themselves, through the global shared memory. This eliminates the system overhead of uselessly copying bulky data from place to place.

There are two special notes to make about there "buf" fields. First, they don't actually have to be used how they're intended to be used. as long as both the client and the server agree on what the contents of these fields are supposed to mean. In this respect, the fields can be used to quickly pass twelve bytes of completely arbitrary information. This is useful because many RPCs only require that a small amount of information be transferred from one process to another, or at least that bulky data be passed in only one direction (like read or write), so that one of the buffer pointers is free to be used quick, tiny data.

Second, on the sending side, the "buffer" that is pointed to does not have to be a "buffer" at all; it can be an arbitrary data structure that has an arbitrary number of pieces, scattered throughout the global memory of the system. The only responsibility of the sender is to insure that no one else will be attempting to modify the shared data simultaneously while the server is accessing it. This scheme is quite ingenious, I think (thank you, thank you). (The scheme may appear to have a security leak in the design, but our system has no real hardware security anyway).

The expected usage of buffers will be for the sender to use near memory for the request and reply buffers and access them as regular near memory to construct and interpret request and reply messages. The receiver will (in the future) access the buffers as far memory (which they may very well be since processes will be allowed to execute on different banks of internal memory and even on different machines), and may wish to fetch parts of messages into near memory for processing. The use of far pointers makes it so that data is copied only when necessary, and copied only once.

4.1.1.3. DATA-CLASS FIELDS

The final four fields, of class "data", are intended to be used to conveniently pass small amounts of arbitrary data. This data can be arbitrary, but the fields do have a convention that should usually be followed, unless both parties agree to an alternative usage.

The "msgOp" field is intended to be the "operation code" that a client process wishes a server to execute. The "msgRet" field is intended to be the return/error code that is returned from the server to the client upon completion of an operation. The "msgObj" field is intended to be used by the client to

indicate which of the server's "objects" the client wishes to perform the operation on. And the "msgData" field is intended to contain four bytes of arbitrary user data that is passed in with an operation and is passed back from the server to give return values. In the spirit of these semantics, the data in all of the fields is send with a request, but only the data in the "msgRet" and "msgData" fields is passed back in a reply operation. None of the other fields are passed back in a reply operation (the field values will remain how they were before the send, for the sender). Take special note that the "msgRepLen" field will not be passed back; if there is less data returned than was asked for by an operation, you will have to encode the "actual" reply-buffer length into the "msgData" field.

4.1.2. SEND

Send() is used to transmit a message to a remote process and get back a reply message. The .AY register contains the near-memory address of the message header, which must have its "msgTo" field filled in to be the pid of the process that the message is being sent to. The sending process will suspend its execution while it is waiting for remote process to process its request. If there is to be bulky reply data for the request (such as there would be for a "read" request to a file server), then space for the reply buffer must be allocated and indicated in the message header. The reply-buffer space should normally be owned by the sender.

If there is an error in passing the message, the the error return will be indicated by the carry flag being set and the error code will be returned in the .A register. Some possible errors will be, in the future: destination process is not valid, and that destination process died before receiving/ replying to your message. (These conditions are not currently checked). Also in the future, this call will work completely transparently for passing messages between machines in a network.

4.1.3. RECEIVE

Receive() is used to receive a message transmitted by a remote process to the current process. The receiver will block until another process does a corresponding Send() operation, and then the message header sent by the sender will be retrieved into the message-header buffer pointed to by the .AY register, for this call. No error returns are possible. The pid of the sending process will be returned in the .AY register as well as in the "msgFrom" field of the receive-message-header buffer. The receiver is then expected to eventually call the Reply() primitive to re-awaken the sender. The receiver is free to do anything it wants to after receiving a message from a process, including receiving messages from other processes. Messages are received from other processes in FIFO order.

A similar ReceiveSpecific() primitive may be provided in the future. It would only accept a message from a specifically named process and would enqueue all other messages that are received before the specific message, to be received later.

4.1.4. REPLY

Reply() is used to re-awaken a process that sent a message that was Receive()d by the current process. The current process is expected to have set up the return information in the reply-message-header buffer and the reply buffer area according to the client's wishes before calling the Reply() primitive. The near address of the reply-message-header buffer is loaded into the .AY register as an argument to the call. Only the "msgFrom", "msgRet", and "msgData" fields need to have values. The "msgFrom" field identifies the process to send the reply message to, and that process must be in the state of waiting for a reply from the Reply()ing process, or an error will be returned. An error is indicated by the carry flag being set on return and the error code is loaded in the .A register. In the case of an error, no action will have been performed by the system.

4.2. IMPLEMENTATION

The fields of the process control block that are used for message passing are restated here:

OFF	SIZ	CLASS	LABEL
12	2	ipc	pcbSendMsgPtr (overlap)
12	2	ipc	pcbRecvMsgPtr (overlap)
14	2	ipc/q	pcbSendQHead
16	2	ipc/q	pcbSendQTail
18	1	ipc/q	pcbSendQFlag
19	1	ipc/q	pcbSendQCount
20	2	ipc	pcbBlockedOn
22	2	ipc	pcbReceiveFrom

The "pcbBlockedOn" field is used to allow Reply() to verify that the pid it is instructed to send a reply message to is indeed waiting for a reply from the task calling Reply(). The "pcbSendQ*" fields constitute a queue head for a list of process control blocks that are waiting to send a message to the current process. The "pcbSendMsgPtr" and "pcbRecvMsgPtr" fields are used to save the message data parameters of a Send() or Receive() call, respectively, when it has to be suspended without a transfer of the message header. When the other process involved performs the corresponding operation, the first process' header buffer pointer is recovered from its process control block. The "pcbReceiveFrom" field is unused at this time.

The process states of STATE_SEND, STATE_REPLY, and STATE_RECEIVE are used with message passing. The STATE_SEND state means that the current process has sent a message to a server process and is waiting for it to do a Receive(). The STATE_REPLY state means that the current process has sent a message to a server process, the message has been Receive()d, and that the current process is waiting for the server process to perform a Reply(). The STATE_RECEIVE state means that the current process has performed a Receive() and is waiting for some other process to perform a corresponding Send(). These state names/meanings may be a bit inconsistent; deal with it.

The implementation of the actual Send(), Receive(), and Reply() operations is actually quite straight-forward. Both Send() and Receive() have to handle two possibile situations: either the other process involved has already performed its corresponding operation and is waiting, or it has not. Reply() is simplified in that it knows that the sender is already waiting for its reply so it can proceed to copy the reply-message-header contents directly.

The Send() primitive (will) checks the given destination pid for validity and then checks the state of the recipient process. If the recipient process is in STATE_RECEIVE, the Send() function copies the message-header contents directly to the receive-header buffer of the recipient. The address of the receive-header buffer is taken from the "pcbRecvMsgPtr" field of the receiver's process control block in this case. The receiver's return value (the sending process' pid) is set up (on the receiving process' stack) and the receiver is awakened while the sender is put to sleep, in STATE_REPLY state (since the receive has already happened, it is waiting for the corresponding Reply()).

If the recipient process is not in the STATE_RECEIVE state, then the sending process will have to wait for the recipient to perform a Receive(). The sender's message-header buffer address is stored into its process control block, the sender's process control block is linked into the recipient process' "pcbSendQ*", and the sender is put to sleep, in the STATE_SEND state.

The Send() function does not set up the return value for the user's system call since that will not be known until another process performs the corresponding Reply(). A return value is set up immediately only in the case of an error. The possible error returns from Send() are: invalid pid and reply too long (in which case the reply is truncated).

The Receive() primitive first checks its "pcbSendQ*" to see if any processes have already tried to send a message to the receiver. If there is a process there, the sender's process control block is removed from the head of the send queue then the sender process' state is changed to STATE_REPLY and the sent message-header contents (dereferenced by the sender's "pcbSendMsgPtr" pointer) are copied into the receiver's message-header buffer. The Receive() primitive then exits returning the pid of the sender. No error returns are possible.

If there is no process enqueued in the recipient process' "pcbSendQ*", then the receiving process is put to sleep in the STATE_RECEIVE state and its message-header buffer pointer is copied into its process control block.

The Reply() primitive verifies that the destination process is valid (but not in the current implementation) and is actually awaiting a reply from the replying process. If not, it craps out. Otherwise, it copies the two message-header fields and awakens the sender. The return value of the sender is (already) set up to be carry-clear (no error) and the Reply() primitive returns error-free too.

The Exit() kernel call does not currently recover from a process performing a Receive() and then Exit()ing before performing the corresponding Reply(). Some care will have to be taken to insure that all process involved in IPC can consistently recover if one of the processes gets blown away, for whatever reason (including Exit()). Such consistent recovery has to be carefully thought out for any kind of operating system; however, since there are only a small number of kernel concepts in this one, consistent recovery is that much easier to insure.

5. CONCLUSION

So there ya have it; the start of a real operating system for the Commodore 128. What the operating system needs in terms of features is to be extended to execute processes on any bank of internal memory, to access far memory, and to be distributed so that it will work across multiple hosts. What it needs in terms of software is: device drivers, a command shell, utility programs, and an assembler that can produce relocatable code. Oh where, oh where shall I ever find such software??? ;-)

APPENDIX A. SOURCE-CODE LISTING

The source code follows. Extract everything between the "-----=-----" lines and save into a file named "bos.s" (or whatever) and then run it through the ACE assembler to generate the executable program (which is also included below for your convenience). The ACE assembler is available for free with the ACE-128/64 system.

I have not gone through and fully documented the source code, since I have been sitting on this program for quite a while and am in a rush to get it out the door. Besides, the functionality of each important component has already been discussed.

-----=-----

```
;simple multitasking kernel by Craig Bruce, started 25-Oct-1994.

;This program is written in the ACE-Assembler format.

   org $1300
   jmp main

;======== declarations ========

pcbNext         = 00 ;(2) mgmt
pcbPrev         = 02 ;(2) mgmt
pcbIsHead       = 04 ;(1) mgmt
pcbQCount       = 05 ;(1) mgmt
pcbSP           = 06 ;(1) ctxt
pcbStackPage    = 07 ;(1) ctxt
pcbZeroPage     = 08 ;(1) ctxt
pcbD506         = 09 ;(1) ctxt
pcbPriority     = 10 ;(1) sche
pcbCountdown    = 11 ;(1) sche
pcbWakeupTime   = 12 ;(2) sche (overlap)
pcbWaitEvent    = 12 ;(1) sche (overlap)
pcbSendMsgPtr   = 12 ;(2) sche (overlap)
pcbRecvMsgPtr   = 12 ;(2) sche (overlap)
pcbSendQHead    = 14 ;(2) ipc
pcbSendQTail    = 16 ;(2) ipc
pcbSendQFlag    = 18 ;(1) ipc
pcbSendQCount   = 19 ;(1) ipc
pcbBlockedOn    = 20 ;(2) ipc
pcbReceiveFrom  = 22 ;(2) ipc
pcbParent       = 24 ;(2) proc
pcbState        = 26 ;(1) proc
pcbSize         = 27

STATE_READY     = $c0
STATE_SEND      = $c1
STATE_RECEIVE   = $c2
STATE_REPLY     = $c3
STATE_DELAY     = $c4
STATE_SUSPENDED = $c5
STATE_EVENT     = $c6

KERN_ERR_OK           = $e0
KERN_ERR_PID_NOT_REPLY = $e1

msgTo           =  0 ;(2)
msgFrom         =  2 ;(2)
msgBuf          =  4 ;(4)
msgLen          =  8 ;(2)
msgRepBuf       = 10 ;(4)
msgRepLen       = 14 ;(2)
msgOp           = 16 ;(1)
msgRet          = 17 ;(1)
msgObj          = 18 ;(2)
msgData         = 20 ;(4)
msgSize         = 24

queueHeadSize   = 6
```

```
nullPcb    : buf pcbSize
delayQueue : buf queueHeadSize
jiffyTime  : buf 2

activePid  = 02 ;(2)
p          = 04 ;(2)
q          = 06 ;(2)
pcbPtr     = 08 ;(2)
msgPtr     = 10 ;(2)
pageAlloc  = 12 ;(1)

;Stack: ($ff)      : exitaddr-1.h
;       ($fe)      : exitaddr-1.l
;       ($fd) sp+07: pc.h
;       ($fc) sp+06: pc.l
;       ($fb) sp+05: status register
;       ($fa) sp+04: .A
;       ($f9) sp+03: .X
;       ($f8) sp+02: .Y
;       ($f7) sp+01: $ff00 save
;       ($f6) sp+00: -empty-

bkBOS    = $0e
bkUser   = $0e
bkSelect = $ff00
vic      = $d000
sid      = $d400
mmuZeroPage  = $d507
mmuStackPage = $d509
IrqExit  = $ff33

; Create  ( .AY=address, .X=priority ) : .AY=pid
; Exit    ( .A=code, .X=$00 )
; MyPid   ( ) : .AY=pid
; MyParentPid ( ) : .AY=parentPid
; Suspend ( )
; Delay   ( .AY=jiffies ) : .CS=err
; Send    ( .AY=msgBuf ) : .CS:.A=err
; Receive ( .AY=msgBuf ) : .AY=senderPid
; Reply   ( .AY=msgBuf[msgRet,msgData] ) : .CS:.A=err

;======== kernel code ========

main = *
   sei
   ;** entry
   lda #bkBOS
   sta bkSelect
   ;** set interrupt vectors
   lda #<IrqHandler
   ldy #>IrqHandler
   sta $0314
   sty $0315
   lda #<BrkHandler
   ldy #>BrkHandler
   sta $0316
   sty $0317
   lda #<NmiHandler
   ldy #>NmiHandler
   sta $0318
   sty $0319
   ;** initialize delay queue
   lda #0
   sta jiffyTime+0
   sta jiffyTime+1
   lda #<delayQueue
   ldy #>delayQueue
   sta q+0
   sty q+1
   jsr QueueInit
   ;** initialize null/boot process
   lda #<nullPcb
   ldy #>nullPcb
   sta nullPcb+pcbNext+0
   sty nullPcb+pcbNext+1
   sta nullPcb+pcbPrev+0
   sty nullPcb+pcbPrev+1
   sta activePid+0
   sty activePid+1
   lda #$ff
```

```
   sta nullPcb+pcbIsHead
   lda #0
   sta nullPcb+pcbQCount
   lda #>$2000
   sta pageAlloc
   lda #STATE_READY
   sta nullPcb+pcbState
   lda #2
   sta nullPcb+pcbPriority
   cli
   jmp Null

Null = *
   ;** create init process
   lda #<Init
   ldy #>Init
   ldx #1
   jsr Create
   ;** go into endless loop
-  inc $0400
   bne +
   inc $0401
   bne +
   inc $0402
   bne +
   inc $0403
+  jmp -

NmiHandler = *
BrkHandler = *
Shutdown = *
   ;** restore interrupt vectors
   sei
   lda #<$fa65
   ldy #>$fa65
   sta $0314
   sty $0315
   lda #<$fa40
   ldy #>$fa40
   sta $0318
   sty $0319
   ldx #250
   txs
   lda #$00
   sta mmuZeroPage
   sta mmuZeroPage+1
   ldx #$01
   stx mmuStackPage
   sta mmuStackPage+1
   lda #%00000100
   sta $d506
   cli
   jmp $4db7

zpSave : buf 1

createAddr     : buf 2
createPriority : buf 1
createZeropage : buf 1
createStack    : buf 1
createPcb      : buf pcbSize

Create = *  ;( .AY=address, .X=priority ) : .AY=pid
   sei
   ;** switch in
   sta createAddr+0
   sty createAddr+1
   stx createPriority
   lda mmuZeroPage
   sta zpSave
   lda #$00
   sta mmuZeroPage
   ;** allocate resources
   lda #$00
   ldy pageAlloc
   sta pcbPtr+0
   sty pcbPtr+1
   iny
   sty createZeropage
   iny
   sty createStack
```

```
   iny
   sty pageAlloc
   cpy #>$c000
   bcc +
   brk   ; recover gracefully from the condition of running out of memory
+
   ;** initialize pcb
   ;** pcbNext          ;(2) mgmt := 0
   ;** pcbPrev          ;(2) mgmt := 0
   ;** pcbIsHead        ;(1) mgmt := 0
   ;** pcbQCount        ;(1) mgmt := 0
   ;** pcbSP            ;(1) ctxt := $f6
   ;** pcbStackPage     ;(1) ctxt := new
   ;** pcbZeroPage      ;(1) ctxt := new
   ;** pcbD506          ;(1) ctxt := $04
   ;** pcbPriority      ;(1) sche := given
   ;** pcbCountdown     ;(1) sche := priority
   ;** pcbWakeupTime    ;(2) sche := 0
   ;** pcbSendQHead     ;(2) ipc  := QueueInit
   ;** pcbSendQTail     ;(2) ipc  := QueueInit
   ;** pcbSendQFlag     ;(1) ipc  := QueueInit
   ;** pcbSendQCount    ;(1) ipc  := QueueInit
   ;** pcbBlockedOn     ;(2) ipc  := 0
   ;** pcbReceiveFrom   ;(2) ipc  := 0
   ;** pcbParent        ;(2) proc := creator
   ;** pcbState         ;(1) proc := STATE_SUSPENDED
   ldx #pcbSize-1
   lda #$00
-  sta createPcb,x
   dex
   bpl -
   lda #$f6
   sta createPcb+pcbSP
   lda createStack
   sta createPcb+pcbStackPage
   lda createZeropage
   sta createPcb+pcbZeroPage
   lda #$04
   sta createPcb+pcbD506
   lda createPriority
   sta createPcb+pcbPriority
   sta createPcb+pcbCountdown
   lda activePid+0
   ldy activePid+1
   sta createPcb+pcbParent+0
   sty createPcb+pcbParent+1
   lda #STATE_SUSPENDED
   sta createPcb+pcbState
   ldy #pcbSize-1
-  lda createPcb,y
   sta (pcbPtr),y
   dey
   bpl -
   lda pcbPtr+0
   clc
   adc #pcbSendQHead
   sta q+0
   lda pcbPtr+1
   adc #0
   sta q+1
   jsr QueueInit

   ;** initialize new stack
   ;** Stack: ($ff)      : exitaddr-1.h     := >ExitAddr
   ;**        ($fe)      : exitaddr-1.l     := <ExitAddr
   ;**        ($fd) sp+07: pc.h             := >Addr
   ;**        ($fc) sp+06: pc.l             := <Addr
   ;**        ($fb) sp+05: status register  := $00
   ;**        ($fa) sp+04: .A               := $00
   ;**        ($f9) sp+03: .X               := $00
   ;**        ($f8) sp+02: .Y               := $00
   ;**        ($f7) sp+01: $ff00 save       := $0e
   ;**        ($f6) sp+00: -empty-
   lda #$00
   ldy createStack
   sta p+0
   sty p+1
   ldy #$f6+1
   lda #bkUser
   sta (p),y  ;$ff00
   iny
```

```
    ldx #4
    lda #$00
-   sta (p),y
    iny
    dex
    bne -
    lda createAddr+0
    sta (p),y
    iny
    lda createAddr+1
    sta (p),y
    iny
    lda #<DefaultExit-1
    sta (p),y
    iny
    lda #>DefaultExit-1
    sta (p),y

    ;** make new process ready
    jsr MakeReady

    ;** switch out
    lda pcbPtr+0
    ldy pcbPtr+1
    ldx zpSave
    stx mmuZeroPage
    clc
    cli
    rts

MakeReady = *  ;( (pcbPtr)=pcb ) ;after activePid
    ldy #pcbState
    lda #STATE_READY
    sta (pcbPtr),y
    lda #<nullPcb
    ldy #>nullPcb
    sta q+0
    sty q+1
    lda activePid+0
    ldy activePid+1
    sta p+0
    sty p+1
    jsr QueueInsert
    rts

QueueInit = *  ;( (q)=queueHead )
    lda q+0
    ldy q+1
    sta queueInitVals+pcbNext+0
    sty queueInitVals+pcbNext+1
    sta queueInitVals+pcbPrev+0
    sty queueInitVals+pcbPrev+1
    lda #$ff
    sta queueInitVals+pcbIsHead
    lda #0
    sta queueInitVals+pcbQCount
    ldy #queueHeadSize-1
-   lda queueInitVals,y
    sta (q),y
    dey
    bpl -
    rts
    queueInitVals : buf queueHeadSize

QueueInsert = *  ;( (q)=queueHead, (p)=nodeToInsertAfter, (pcbPtr)=newItem )
    ;** q->count +:= 1
    clc
    ldy #pcbQCount
    lda (q),y
    adc #1
    sta (q),y

    ;** pcbPtr->next := p->next
    ldy #pcbNext
    lda (p),y
    sta (pcbPtr),y
    iny
    lda (p),y
    sta (pcbPtr),y

    ;** pcbPtr->prev := p
```

```
   iny
   lda p+0
   sta (pcbPtr),y
   iny
   lda p+1
   sta (pcbPtr),y

   ;** p->next->prev := pcbPtr
   ldy #pcbNext
   lda (p),y
   sta q+0
   iny
   lda (p),y
   sta q+1
   ldy #pcbPrev
   lda pcbPtr+0
   sta (q),y
   iny
   lda pcbPtr+1
   sta (q),y

   ;** p->next := pcbPtr
   ldy #pcbNext
   lda pcbPtr+0
   sta (p),y
   iny
   lda pcbPtr+1
   sta (p),y
   rts

QueueUnlink = *  ;( (q)=queueHead, (pcbPtr)=node )  ;uses p
   ;** pcbPtr->next->prev := pcbPtr->prev
   ldy #pcbNext
   lda (pcbPtr),y
   sta p+0
   iny
   lda (pcbPtr),y
   sta p+1
   ldy #pcbPrev
   lda (pcbPtr),y
   sta (p),y
   iny
   lda (pcbPtr),y
   sta (p),y

   ;** pcbPtr->prev->next := pcbPtr->next
   ldy #pcbPrev
   lda (pcbPtr),y
   sta p+0
   iny
   lda (pcbPtr),y
   sta p+1
   ldy #pcbNext
   lda (pcbPtr),y
   sta (p),y
   iny
   lda (pcbPtr),y
   sta (p),y

   ;** q->count -:= 1
   ldy #pcbQCount
   lda (q),y
   sec
   sbc #1
   sta (q),y
   rts

IrqHandler = *
   cld
   lda #bkBOS
   sta bkSelect
   lda vic+$19
   bpl +
   and #1
   bne Sixty
+  lda $dc0d

Sixty = *
   sta vic+$19
   ;** save full context
   lda mmuZeroPage
```

```
   ldx #$00
   stx mmuZeroPage
   ldy #pcbZeroPage
   sta (activePid),y
   ldy #pcbSP
   tsx
   txa
   sta (activePid),y
   ldy #pcbStackPage
   lda mmuStackPage
   sta (activePid),y
   ldy #pcbD506
   lda $d506
   sta (activePid),y

   ;** process interrupt
   inc jiffyTime+0
   bne +
   inc jiffyTime+1
+  lda delayQueue+pcbQCount
   beq +
   jsr DelayIrqAwake
+  nop

   ;** select new process
-  ldy #pcbPriority   ;give cur full count
   lda (activePid),y
   iny
   sta (activePid),y
   beq ++
-  ldy #pcbNext       ;find next proc
   lda (activePid),y
   tax
   iny
   lda (activePid),y
   stx activePid+0
   sta activePid+1
ExitKernel = *
   ldy #pcbCountdown
   lda (activePid),y
   beq ++
   sec
   sbc #1
   sta (activePid),y
   beq +
   jmp -
+  ;check if null process
   ldy #pcbIsHead
   lda (activePid),y
   bpl +
   iny
   lda (activePid),y  ;only run null if only proc
   bne --
+  ;we've got a winner

   ;** restore full context and exit
   ldy #pcbD506
   lda (activePid),y
   sta $d506
   ldy #pcbStackPage
   lda (activePid),y
   sta mmuStackPage
   ldy #pcbSP
   lda (activePid),y
   tax
   txs
   ldy #pcbZeroPage
   lda (activePid),y
   sta mmuZeroPage
   jmp IrqExit

DefaultExit = *
   lda #$00
   ldx #$00
Exit = *  ;( .A=code, .X=$00 )
   jmp Suspend
   brk

MyPid = *  ;( ) : .AY=pid
   lda #$00
   ldx mmuZeroPage
```

```
   sta mmuZeroPage
   lda activePid+0
   ldy activePid+1
   stx mmuZeroPage
   clc
   rts

MyParentPid = *  ;( ) : .AY=parentPid
   lda #$00
   ldx mmuZeroPage
   sta mmuZeroPage
   ldy #pcbParent
   lda (activePid),y
   pha
   iny
   lda (activePid),y
   tay
   pla
   stx mmuZeroPage
   clc
   rts

enterKernSave : buf 4

EnterKernel = *
   ;** set up process stack as if it had performed an interrupt
   ;** necessary if process will block
   ;** called as a one-level-deep subroutine of the system call
   sta enterKernSave+2
   ;** save system-call return address
   pla
   sta enterKernSave+0
   pla
   sta enterKernSave+1
   ;** increment user-process return address (rts -> rti)
   pla
   clc
   adc #1
   sta enterKernSave+3
   pla
   adc #0
   pha
   lda enterKernSave+3
   pha
   ;** set up processor registers as-is, status $00
   lda #$00
   pha
   lda enterKernSave+2
   pha
   txa
   pha
   tya
   pha
   lda $ff00  ;xxx change for multi-banks
   pha
   ;** save info into pcb
   lda mmuZeroPage
   ldx #$00
   stx mmuZeroPage
   ldy #pcbZeroPage
   sta (activePid),y
   dey
   lda mmuStackPage
   sta (activePid),y
   dey
   tsx
   txa
   sta (activePid),y
   ldy #pcbD506
   lda $d506
   sta (activePid),y
   ;** restore system-call return address
   ;** (continue to use user-process stack)
   lda enterKernSave+1
   pha
   lda enterKernSave+0
   pha
   rts

Suspend = *  ;( )    ;suspend self
   sei
```

```
   jsr EnterKernel
   jsr SuspendSub
   jmp ExitKernel

SuspendSub = *  ;( activePid ) : activePid, pcbPtr, q=nullPcb
   ;** Remove the active pid from the ready queue and set another pid to
   ;** active; set pcbPtr to point to the suspended process; and set the
   ;** process state to "suspended".
   lda activePid+0
   sta pcbPtr+0
   lda activePid+1
   sta pcbPtr+1
   lda #<nullPcb
   ldy #>nullPcb
   sta q+0
   sty q+1
   jsr QueueUnlink
   ldy #pcbNext
   lda (pcbPtr),y
   sta activePid+0
   iny
   lda (pcbPtr),y
   sta activePid+1
   ldy #pcbState
   lda #STATE_SUSPENDED
   sta (pcbPtr),y
   rts

Delay = *  ;( .AY=jiffies ) : .CS=err
   cmp #0
   bne +
   cpy #0
   bne +
   clc
   rts
+  sei
   sta delayTime+0
   sty delayTime+1
   jsr EnterKernel
   jsr SuspendSub
   ldy #pcbState
   lda #STATE_DELAY
   ldy #pcbWakeupTime
   clc
   lda delayTime+0
   adc jiffyTime+0
   sta delayTime+0
   sta (pcbPtr),y
   iny
   lda delayTime+1
   adc jiffyTime+1
   sta delayTime+1
   sta (pcbPtr),y
   lda #0
   rol
   sta delayTime+2
   lda #<delayQueue
   ldy #>delayQueue
   sta q+0
   sty q+1
   sta p+0
   sty p+1
   jsr DelayFindSpot
   jsr QueueInsert
   jmp ExitKernel
   delayTime : buf 3
   pTimeHi   : buf 1

DelayFindSpot = *  ;( (q)=queue, (p)=queueHead, (pcbPtr) ) : p=prevNode
   jsr IncPtrP
   ldy #pcbIsHead
   lda (p),y
   bne DelayFindSpotExit
   ldy #pcbWakeupTime
   lda (p),y
   cmp jiffyTime+0
   iny
   lda (p),y
   sbc jiffyTime+1
   ldx #0
   bcs +
```

```
   inx
+  stx pTimeHi
   dey
   lda delayTime+0
   cmp (p),y
   iny
   lda delayTime+1
   sbc (p),y
   lda delayTime+2
   sbc pTimeHi
   bcs DelayFindSpot

   DelayFindSpotExit = *
   ;xx fall through

DecPtrP = *  ;( (p) ) : (p):=(p)->prev
   ldy #pcbPrev
   lda (p),y
   tax
   iny
   lda (p),y
   stx p+0
   sta p+1
   rts

IncPtrP = *  ;( (p) ) : (p):=(p)->next
   ldy #pcbNext
   lda (p),y
   tax
   iny
   lda (p),y
   stx p+0
   sta p+1
   rts

DelayIrqAwake = *
   lda delayQueue+pcbNext+0
   ldy delayQueue+pcbNext+1
   sta pcbPtr+0
   sty pcbPtr+1
   ldy #pcbWakeupTime
   lda (pcbPtr),y
   cmp jiffyTime+0
   beq +
   rts
+  iny
   lda (pcbPtr),y
   cmp jiffyTime+1
   beq +
   rts
+  lda #<delayQueue
   ldy #>delayQueue
   sta q+0
   sty q+1
   jsr QueueUnlink
   jsr MakeReady
   jmp DelayIrqAwake

msgPtrSave : buf 2

Send = *  ;( .AY=msgBuf ) : .CS:.A=err
   sei
   sta msgPtrSave+0
   sty msgPtrSave+1
   jsr EnterKernel
   jsr SuspendSub
   lda msgPtrSave+0
   ldy msgPtrSave+1
   sta msgPtr+0
   sty msgPtr+1
   ldy #msgTo
   lda (msgPtr),y
   sta q+0
   iny
   lda (msgPtr),y
   sta q+1
   ;xx should verify that receiver is a process
   ldy #pcbSendMsgPtr
   lda msgPtr+0
   sta (pcbPtr),y
   iny
```

```
   lda msgPtr+1
   sta (pcbPtr),y
   ldy #pcbBlockedOn
   lda q+0
   sta (pcbPtr),y
   iny
   lda q+1
   sta (pcbPtr),y
   ldy #pcbState
   lda (q),y
   cmp #STATE_RECEIVE
   beq SendToReceiverBlocked
   lda #STATE_SEND
   sta (pcbPtr),y
   clc
   lda q+0
   adc #pcbSendQHead
   sta q+0
   bcc +
   inc q+1
+  ldy #pcbPrev
   lda (q),y
   sta p+0
   iny
   lda (q),y
   sta p+1
   jsr QueueInsert
   jmp ExitKernel

SendToReceiverBlocked = *
   lda #STATE_REPLY
   sta (pcbPtr),y
   ldy #pcbRecvMsgPtr
   lda (q),y
   sta p+0
   iny
   lda (q),y
   sta p+1
   jsr CopyMessage
   lda pcbPtr+0
   ldy pcbPtr+1
   ldx q+0
   stx pcbPtr+0
   ldx q+1
   stx pcbPtr+1
   ldx #$00
   clc
   jsr SetReturn
   jmp MakeReady
   jmp ExitKernel

setretSave : buf 4

SetReturn = *  ;( (pcbPtr)=proc, .AXY=regvals, .C=cval ) : (p)=junk
   sta setretSave+2
   stx setretSave+1
   sty setretSave+0
   php
   pla
   and #$01
   sta setretSave+3
   ldy #pcbStackPage
   lda (pcbPtr),y
   sta p+1
   ldy #pcbSP
   lda (pcbPtr),y
   clc
   adc #2
   sta p+0
   ldy #3
-  lda setretSave,y
   sta (p),y
   dey
   bpl -
   rts

CopyMessage = *  ;( (pcbPtr)=sender, (msgPtr)=sendmsg, (p)=recvmsg )
   ldy #msgFrom
   lda pcbPtr+0
   sta (msgPtr),y
   iny
```

```
   lda pcbPtr+1
   sta (msgPtr),y
   ldy #msgSize-1
-  lda (msgPtr),y
   sta (p),y
   dey
   bpl -
   rts

Receive = *  ;( .AY=msgBuf ) : .AY=senderPid
   sei
   sta msgPtrSave+0
   sty msgPtrSave+1
   lda mmuZeroPage
   pha
   lda #$00
   sta mmuZeroPage
   ldy #pcbSendQCount
   lda (activePid),y
   bne ReceiveFromSender
   pla
   sta mmuZeroPage
   jsr EnterKernel
   jsr SuspendSub
   lda #STATE_RECEIVE
   sta (pcbPtr),y
   ldy #pcbRecvMsgPtr
   lda msgPtrSave+0
   sta (pcbPtr),y
   iny
   lda msgPtrSave+1
   sta (pcbPtr),y
   jmp ExitKernel

ReceiveFromSender = *  ;( (activePid), (msgPtrSave) )
   lda activePid+0
   ldy activePid+1
   clc
   adc #pcbSendQHead
   bcc +
   iny
+  sta q+0
   sty q+1
   ldy #pcbSendQHead
   lda (activePid),y
   sta pcbPtr+0
   iny
   lda (activePid),y
   sta pcbPtr+1
   jsr QueueUnlink  ;( (q)=queueHead, (pcbPtr)=node )  ;uses p
   ldy #pcbSendMsgPtr
   lda (pcbPtr),y
   sta msgPtr+0
   iny
   lda (pcbPtr),y
   sta msgPtr+1
   lda msgPtrSave+0
   ldy msgPtrSave+1
   sta p+0
   sty p+1
   jsr CopyMessage  ;( (pcbPtr)=sender, (msgPtr)=sendmsg, (p)=recvmsg )
   ldy #pcbState
   lda #STATE_REPLY
   sta (pcbPtr),y
   ldx pcbPtr+0
   ldy pcbPtr+1
   pla
   sta mmuZeroPage
   txa
   cli
   clc
   rts

zpPtrSave : buf 1

Reply = *  ;( .AY=msgBuf[msgRet,msgData] ) : .CS:.A=err
   sei
   ;** switch to kernel
   ldx mmuZeroPage
   stx zpPtrSave
   ldx #$00
```

```
   stx mmuZeroPage
   sta msgPtr+0
   sty msgPtr+1
   ;** find and check the sender
   ldy #msgFrom
   lda (msgPtr),y
   sta pcbPtr+0
   iny
   lda (msgPtr),y
   sta pcbPtr+1
   ;xx verify that receiver is a pcb here
   ldy #pcbState
   lda (pcbPtr),y
   cmp #STATE_REPLY
   beq +
-  lda #KERN_ERR_PID_NOT_REPLY
   ldx zpPtrSave
   stx mmuZeroPage
   sec
   cli
   rts
+  ldy #pcbBlockedOn
   lda (pcbPtr),y
   cmp activePid+0
   bne -
   iny
   lda (pcbPtr),y
   cmp activePid+1
   bne -
   ;** copy the reply contents
   ldy #pcbSendMsgPtr
   lda (pcbPtr),y
   sta p+0
   iny
   lda (pcbPtr),y
   sta p+1
   ldy #msgRet
   lda (msgPtr),y
   sta (p),y
   ldy #msgData
-  lda (msgPtr),y
   sta (p),y
   iny
   cpy #msgData+4
   bcc -
   ;** wake up the sender and exit
   jsr MakeReady
   ldx zpPtrSave
   stx mmuZeroPage
   clc
   cli
   rts

;======== test application ========

testNumber : buf 1

Init = *
   lda #1
   sta testNumber
   lda #<TestSid1
   ldy #>TestSid1
   ldx #2
   jsr Create
   lda #<TestDelay1
   ldy #>TestDelay1
   ldx #1
   jsr Create
   lda #<TestDelay2
   ldy #>TestDelay2
   ldx #1
   jsr Create
   lda #<TestDelay3
   ldy #>TestDelay3
   ldx #1
   jsr Create
   lda #<TestDelay4
   ldy #>TestDelay4
   ldx #1
   jsr Create
   lda #<TestDelay5
```

```
   ldy #>TestDelay5
   ldx #1
   jsr Create
   lda #<Blabber1
   ldy #>Blabber1
   ldx #1
   jsr Create
   lda #<Spinner1
   ldy #>Spinner1
   ldx #1
   jsr Create
   jmp KernelServer

TestSid1 = *
   ldx #$1c-1
   lda #$00
-  sta $d400,x
   dex
   bpl -
   lda #$50
   sta 2
   sta 3
   lda #$08
   sta $d418
   lda #$00
   ldy #$08
   sta $d402
   sty $d403
   lda #$41
   sta $d404
   lda #$00
   sta $d405
   lda #$f0
   sta $d406
-  lda 2
   ldy 3
   sta $d400
   sty $d401
   lda 2
   ora 3
   bne +
   lda #120
   ldy #0
   jsr Delay
+  inc 2
   bne +
   inc 3
+  inc $d020
   tsx
   jmp -

TestDelay1 = *
   jsr MyParentPid
   sta testDelay1Msg+msgTo+0
   sty testDelay1Msg+msgTo+1
   lda #<testDelay1Txt
   ldy #>testDelay1Txt
   sta testDelay1Msg+msgBuf+0
   sty testDelay1Msg+msgBuf+1
-  lda #<60
   ldy #>60
   jsr Delay
   inc $581
   lda #<testDelay1Msg
   ldy #>testDelay1Msg
   jsr Send
   jmp -
   testDelay1Txt : db "Hi, this is delay process 1 *\n",0

TestDelay2 = *
   jsr MyParentPid
   sta testDelay2Msg+msgTo+0
   sty testDelay2Msg+msgTo+1
   lda #<testDelay2Txt
   ldy #>testDelay2Txt
   sta testDelay2Msg+msgBuf+0
   sty testDelay2Msg+msgBuf+1
-  lda #<120
   ldy #>120
   jsr Delay
   inc $582
```

```
   lda #<testDelay2Msg
   ldy #>testDelay2Msg
   jsr Send
   jmp -
   testDelay2Txt : db "Hi, this is delay process 2\n",0

TestDelay3 = *
   jsr MyParentPid
   sta testDelay3Msg+msgTo+0
   sty testDelay3Msg+msgTo+1
   lda #<testDelay3Txt
   ldy #>testDelay3Txt
   sta testDelay3Msg+msgBuf+0
   sty testDelay3Msg+msgBuf+1
-  lda #<180
   ldy #>180
   jsr Delay
   inc $583
   lda #<testDelay3Msg
   ldy #>testDelay3Msg
   jsr Send
   jmp -
   testDelay3Txt : db "Hi, this is delay process 3\n",0

TestDelay4 = *
   jsr MyParentPid
   sta testDelay4Msg+msgTo+0
   sty testDelay4Msg+msgTo+1
   lda #<testDelay4Txt
   ldy #>testDelay4Txt
   sta testDelay4Msg+msgBuf+0
   sty testDelay4Msg+msgBuf+1
-  lda #<240
   ldy #>240
   jsr Delay
   inc $584
   lda #<testDelay4Msg
   ldy #>testDelay4Msg
   jsr Send
   jmp -
   testDelay4Txt : db "Hi, this is delay process 4\n",0

TestDelay5 = *
   jsr MyParentPid
   sta testDelay5Msg+msgTo+0
   sty testDelay5Msg+msgTo+1
   lda #<testDelay5Txt
   ldy #>testDelay5Txt
   sta testDelay5Msg+msgBuf+0
   sty testDelay5Msg+msgBuf+1
-  lda #<300
   ldy #>300
   jsr Delay
   inc $585
   lda #<testDelay5Msg
   ldy #>testDelay5Msg
   jsr Send
   jmp -
   testDelay5Txt : db "Hi, this is delay process 5\n",0

Blabber1 = *
   jsr MyParentPid
   sta blabber1Msg+msgTo+0
   sty blabber1Msg+msgTo+1
   lda #<blabber1Txt
   ldy #>blabber1Txt
   sta blabber1Msg+msgBuf+0
   sty blabber1Msg+msgBuf+1
-  inc $580
   lda #<blabber1Msg
   ldy #>blabber1Msg
   jsr Send
   jmp -
   blabber1Txt : db "Hi, this is blabber\n",0

Spinner1 = *
   jsr MyParentPid
   sta spinner1Msg+msgTo+0
   sty spinner1Msg+msgTo+1
   lda #<spinner1Txt
   ldy #>spinner1Txt
```

```
   sta spinner1Msg+msgBuf+0
   sty spinner1Msg+msgBuf+1
-  inc $580
   lda #<spinner1Msg
   ldy #>spinner1Msg
   jsr Send
   jmp -
   spinner1Txt : db "Hi, this is spinner +\n",0

KernelServer = *
   lda #$00
   sta mmuZeroPage
   lda #14
   jsr $ffd2
-  lda #<ksMsg
   ldy #>ksMsg
   jsr Receive
   lda ksMsg+msgBuf+0
   ldy ksMsg+msgBuf+1
   sta $80
   sty $81
   ldy #0
-  lda ($80),y
   beq +
   jsr $ffd2
   iny
   bne -
+  lda #<ksMsg
   ldy #>ksMsg
   jsr Reply
   jmp --

bss = *
testDelay1Msg = $c00  ;** put these here to save pgm memory
testDelay2Msg = testDelay1Msg+msgSize
testDelay3Msg = testDelay2Msg+msgSize
testDelay4Msg = testDelay3Msg+msgSize
testDelay5Msg = testDelay4Msg+msgSize
blabber1Msg   = testDelay5Msg+msgSize
spinner1Msg   = blabber1Msg+msgSize
ksMsg         = spinner1Msg+msgSize
-----=-----
```

APPENDIX B. UUENCODED DEMO PROGRAM

The uuencoded demo system follows. You can extract it with any uudecoder or with version 2.00 or higher of "unbcode" (ACE has only version 1.00).

```
-nucode-begin 1 bos
begin 640 bos
M`!-,)A,````````````````````````````````````````````'BI#HT`
M_ZEVH!6-%`.,%0.IJZ`3C18#C!<#J:N@$XT8`XP9`ZD`C203C243J1Z@$X4&
MA`<@U12I`Z`3C0,3C`03C043C`83A0*$`ZG_C0<3J0"-"!.I((4,J<"-'1.I
M`HT-$UA,C1.I.Z`9H@$@_1/N``30#>X!!-`([@($T`/N`P1,EA-XJ66@^HT4
M`XP5`ZE`H/J-&`.,&0.B^IJI`(T'U8T(U:(!C@G5C0K5J02-!M583+=-````
M`````````````````````````````````>(W=$XS>$X[?$ZT'U8W<
M$ZD`C0?5J0"D#(4(A`G(C.`3R(SA$\B$#,#`D`$`HAJI`)WB$\H0^JGVC>@3
MK>$3C>D3K>`3C>H3J02-ZQ.MWQ.-[!.-[1.E`J0#C?H3C/L3J<6-_!.@&KGB
M$Y$(B!#XI0@8:0Z%!J4):0"%!R#5%*D`K.$3A02$!:#WJ0Z1!,BB!*D`D03(
MRM#ZK=T3D03(K=X3D03(J0F1!,BI%I$$(+L4I0BD":[<$XX'U1A88*`:J<"1
M"*D#H!.%!H0'I0*D`X4$A`4@`!5@I0:D!XWZ%(S[%(W\%(S]%*G_C?X4J0"-
M_Q2@!;GZ%)$&B!#X8`````````!B@!;$&:0&1!J``L021",BQ!)$(R*4$D0C(
MI061"*``L02%!LBQ!(4'H`*E")$&R*4)D0:@`*4(D03(I0F1!&"@`+$(A03(
ML0B%!:`"L0B1!,BQ")$$H`*Q"(4$R+$(A06@`+$(D03(L0B1!*`%L08XZ0&1
M!F#8J0Z-`/^M&=`0!"D!T`.M#=R-&="M!]6B`(X'U:`(D0*@!KJ*D0*@!ZT)
MU9$"H`FM!M61`NXD$]`#[B43K2,3\`,@71?JH`JQ`LB1`O`GH`"Q`JK(L0*&
M`H4#H`NQ`O`5..D!D0+P`TS%%:`$L0(0!<BQ`M#0H`FQ`HT&U:``L0*-"=6@
M!K$"JIJ@"+$"C0?53#/_J0"B`$R.%@"I`*X'U8T'U:4"I`..!]488*D`K@?5
MC0?5H!BQ`DC(L0*H:(X'U1A@`````(T\%FB-.A9HC3L6:!AI`8T]%FAI`$BM
M/19(J0!(K3P62(I(F$BM`/](K0?5H@".!]6@")$"B*T)U9$"B+J*D0*@":T&
MU9$"K3L62*TZ%DA@>"`^%B"8%DS1%:4"A0BE`X4)J0.@$X4&A`<@0!6@`+$(
MA0+(L0B%`Z`:J<61"&#)`-`&P`#0`AA@>(T-%XP.%R`^%B"8%J`:J<2@#!BM
M#1=M)!.-#1>1",BM#A=M)1.-#A>1"*D`*HT/%ZD>H!.%!H0'A02$!2`1%R``
M%4S1%0`````@4!>@!+$$T"F@#+$$S203R+$$[243H@"P`>B.$!>(K0T7T03(
MK0X7\02M#Q?M$!>PSJ`"L02JR+$$A@2%!6"@`+$$JLBQ!(8$A05@K1X3K!\3
MA0B$":`,L0C-)!/P`6#(L0C-)1/P`6"I'J`3A0:$!R!`%2"[%$Q=%P``>(V+
M%XR,%R`^%B"8%JV+%ZR,%X4*A`N@`+$*A0;(L0J%!Z`,I0J1",BE"Y$(H!2E
M!I$(R*4'D0B@&K$&R<+P(*G!D0@8I09I#H4&D`+F!Z`"L0:%!,BQ!H4%(``5
M3-$5J<.1"*`,L0:%!,BQ!H4%($48I0BD":8&A@BF!X8)H@`8(!L8(+L43-$5
M`````(T9&(X8&(P7&`AH*0&-&AB@![$(A06@!K$(&&D"A02@`[D7&)$$B!#X
M8*`"I0B1"LBE"9$*H!>Q"I$$B!#Y8`B-BQ>,C!>M!]5(J0"-!]6@$[$"T!YH
```

```
MC0?5(#X6()@6J<*1"*`,K8L7D0C(K8P7D0A,T16E`J0#&&D.D`'(A0:$!Z`.
ML0*%",BQ`H4)($`5H`RQ"(4*R+$(A0NMBQ>LC!>%!(0%($48H!JIPY$(I@BD
M"6B-!]6*6!A@`'BN!]6.U!BB`(X'U84*A`N@`K$*A0C(L0J%":`:L0C)P_`+
MJ>&NU!B.!]4X6&"@%+$(Q0+0[<BQ",4#T.:@#+$(A03(L0B%!:`1L0J1!*`4
ML0J1!,C`&)#W(+L4KM08C@?5&%A@`*D!C3H9J8N@&:("(/T3J=V@&:(!(/T3
MJ2.@&J(!(/T3J6>@&J(!(/T3J:N@&J(!(/T3J>^@&J(!(/T3J3.@&Z(!(/T3
MJ6B@&Z(!(/T33)\;HANI`)T`U,H0^JE0A0*%`ZD(C1C4J0"@"(T"U(P#U*E!
MC034J0"-!=2I\(T&U*4"I`.-``-2,`=2E`@4#T`>I>*``(+T6Y@+0`N8#[B#0
MNDRY&2`C%HT`#(P!#*D$H!J-!`R,!0RI/*``(+T6[H$%J0"@#""-%TSP&<A)
M+"!42$E3($E3($1%3$%9(%!23T-%4U,@,2`J#0`@(Q:-&`R,&0RI2J`:C1P,
MC!T,J7B@`""]%NZ"!:D8H`P@C1=,-AK(22P@5$A)4R!)4R!$14Q!62!04D]#
M15-3(#(-`"`C%HTP#(PQ#*F.H!J--`R,-0RIM*``(+T6[H,%J3"@#""-%TQZ
M&LA)+"!42$E3($E3($1%3$%9(%!23T-%4U,@,PT`(",6C4@,C$D,J=*@&HU,
M#(Q-#*GPH``@O1;NA`6I2*`,((T73+X:R$DL(%1(25,@25,@1$5,05D@4%)/
M0T534R`T#0`@(Q:-8`R,80RI%J`;C60,C&4,J2R@`2"]%NZ%!:E@H`P@C1=,
M`AO(22P@5$A)4R!)4R!$14Q!62!04D]#15-3(#4-`"`C%HUX#(QY#*E3H!N-
M?`R,?0SN@`6I>*`,((T73$8;R$DL(%1(25,@25,@0DQ!0D)%4@T`(",6C9`,
MC)$,J8B@&XV4#(R5#.Z`!:F0H`P@C1=,>QO(22P@5$A)4R!)4R!34$E.3D52
M("L-`*D`C0?5J0X@TO^IJ*`,(%H8K:P,K*T,A8"$@:``L8#P!B#2_\C0]JFH
(H`P@U1A,J1L`

end
-nucode-end 1 2258 1430bdc2

================================================================================END===
```